Containers and Kubernetes Security

Alessandro Brighente

Università degli Studi di Padova

SPRITZ Security & Privacy Research Group

Sunburst

BBC Home News Sport Reel ...

NEWS

Home | War in Ukraine | Climate | Video | World | UK | Bu

Tech | Science | Entertainment & Arts

Tech

US cyber-attack: US energy department confirms it was hit by Sunburst hack

SolarWinds: Why the Sunburst hack is so serious

What is a Container?

- A standard unit of software that packages up code and all of its dependencies so that the applications runs quickly and reliably from one computing environment to another
- It bundles an application’s code together with the related configuration files, libraries, and dependencies required for the app to run
- Allow for deployment of applications seamlessly across environments

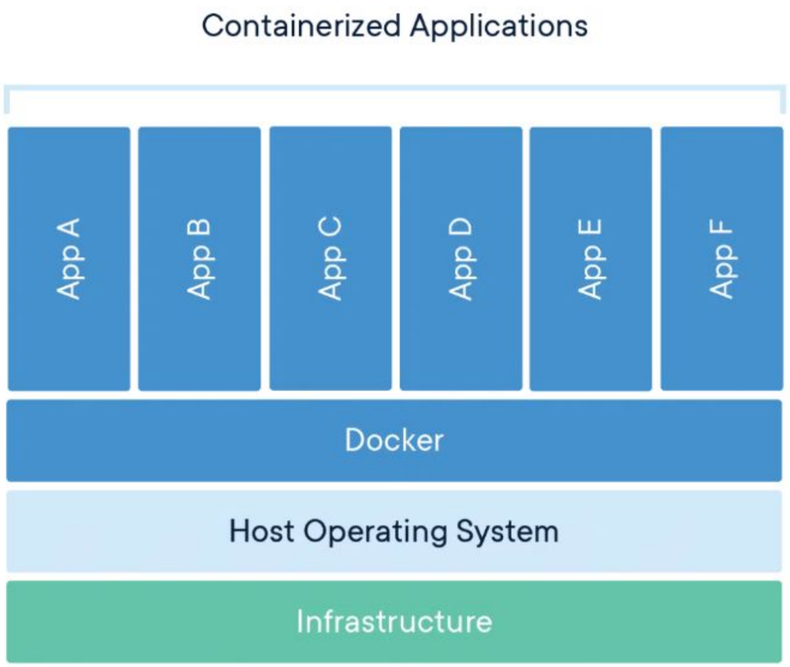
Container vs. Virtual Machine

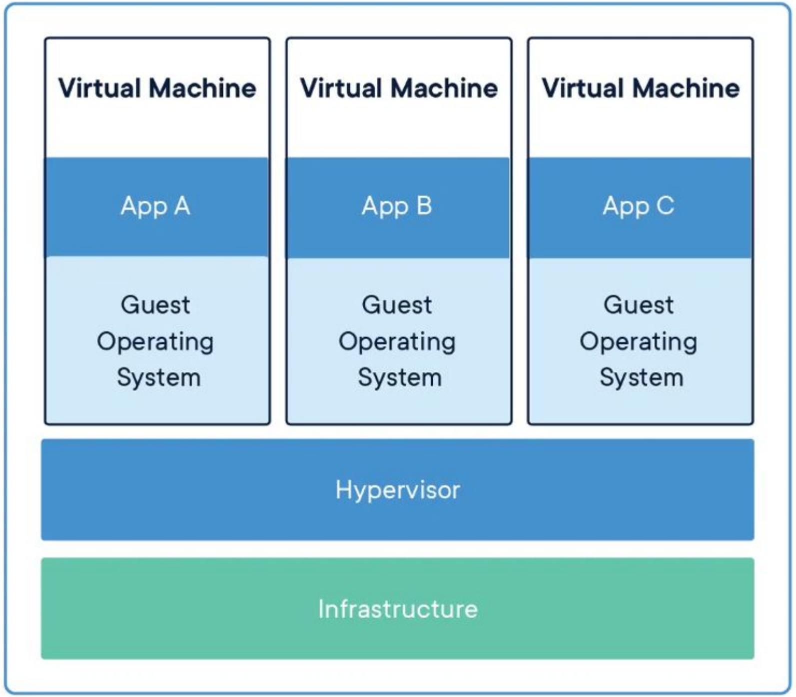
- **Virtual machines** virtualize the underlying hardware so that multiple OSs instances can run on the hardware
- Each VM runs an OS and has access to the virtualized resources representing the underlying hardware
- They allow to run different OSs on the same server, efficient and cost-effective utilization of hardware resources, faster server provisioning
- Drawbacks: each VM contains an OS image, libraries, applications,.. Can become quite large

Container vs. Virtual Machine

- **Container** virtualizes the underlying OS and makes the containerized app believe that the OS and the underlying resources (CPU, memory, file storage,..) belong to it
- Since the differences in underlying OS and infrastructure are abstracted, the container can deployed and run everywhere
- More efficient and lightweight than VMs, as they do not need to bring their own OS and libraries
- They are however constrained to the OS they are defined for

Container vs. Virtual Machine





Containers in Cloud

- **Containers can be used for Cloud-Native Applications**
- They rely on containers for a common operational model across environments (public, private, hybrid)
- The goal is to increase software delivery velocity and service reliability
- The portability and flexibility of containers make them ideal for building *microservices architectures*
- You may benefit from *orchestrators* when needing to run a large number of containers

Docker

- Containers existed for several years before Docker, but Docker's easy to use command line tools made this technology explode in 2013
- You can run multiple side-by-side containers without them interfering one another
- Problem solved: dependencies are isolated and you can have containers that use different versions of a package on the same machine

Containers Threat Model

- Let's consider the actors involved
  - External attackers
  - Internal attackers
  - Malicious internal actors
  - Inadvertent internal actors (accidentally cause problem)
  - Applications processes trying to compromise your system
- Permissions
  - Access to user accounts?
  - What permissions do they have on the system? RBAC
  - What network access do they have? Virtual private cloud?

Containers Threat Model

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

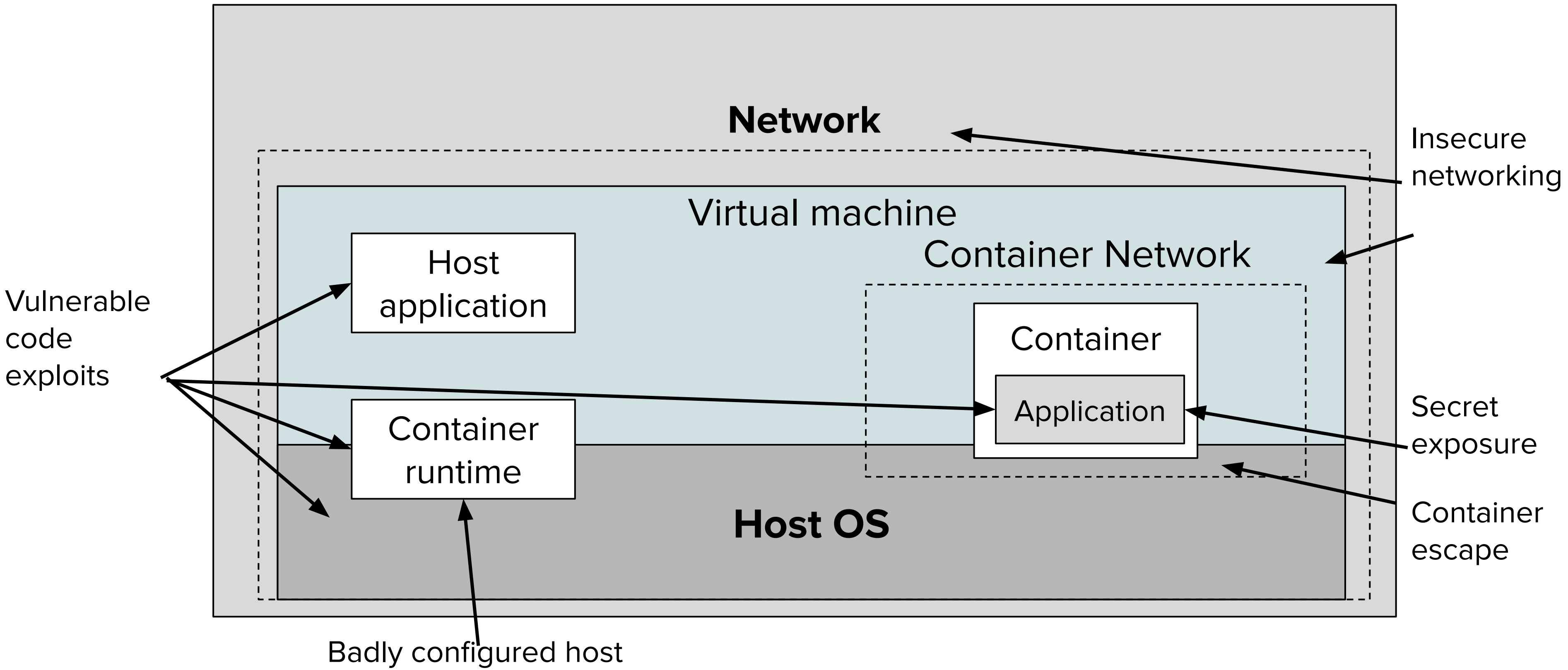
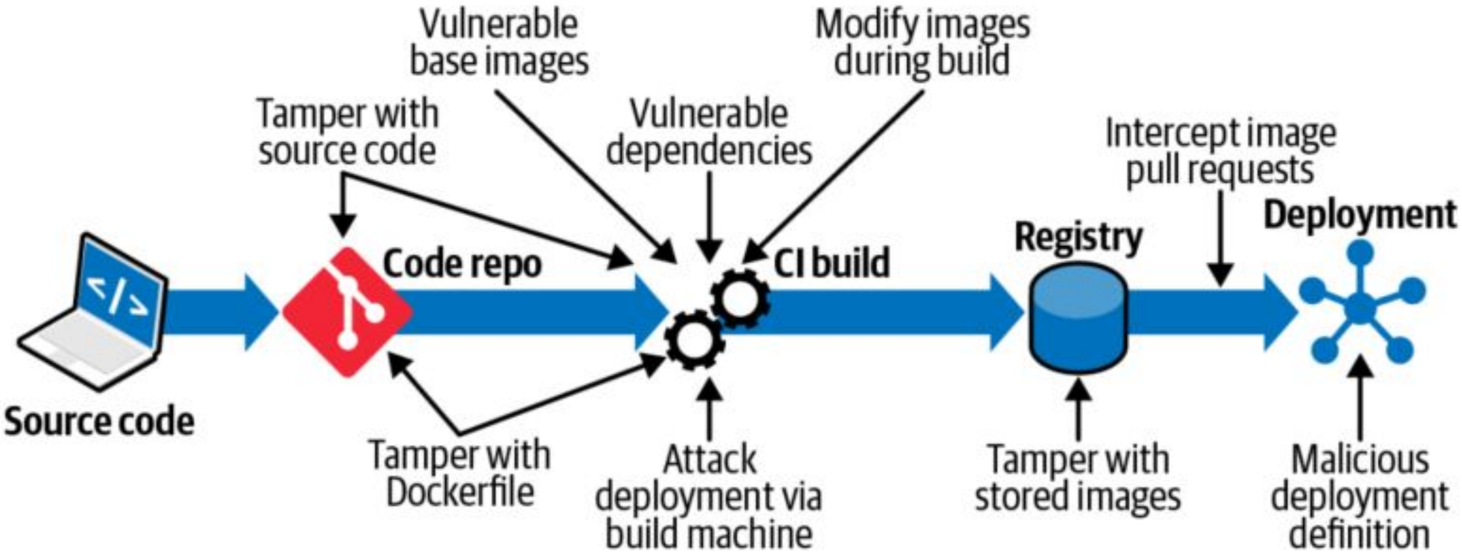


Image Security

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- Main point: ensure that the intended images are what gets used



Linux System Calls

- Applications run in the *user space*, having a lower level of privilege than the operating system kernel
- If an application wants to access a file, use the network, or get the time, it should ask the kernel to do so
- The programmatic interface that the user space code uses to make these requests is known as the *system call* or *syscall* interface
- In a Linux OS, there are more than 300 syscalls
- Applications use syscall in the same way both inside and outside the container environment

Setuid and setgid

- When you execute a file, the process that gets started inherits your user ID

```
vagrant@vagrant:~$ ls -l `which ping`
-rwsr-xr-x 1 root root 64424 Jun 28 11:05 /bin/ping
vagrant@vagrant:~$ cp /bin/ping ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwxr-xr-x 1 vagrant vagrant 64424 Nov 24 18:51 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
```

Setuid and setgid

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- We can change the ownership, but still cannot run it unless root

```
vagrant@vagrant:~$ sudo chown root ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwxr-xr-x 1 root vagrant 64424 Nov 24 18:55 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
```

Setuid and setgid

- If we set the UID bit...

```
vagrant@vagrant:~$ sudo chmod +s ./myping
vagrant@vagrant:~$ ls -l ./myping
-rwsr-sr-x 1 root vagrant 64424 Nov 24 18:55 ./myping
vagrant@vagrant:~$ ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2052ms
```

Security implications of setuid

- Imagine setting the setuid bit on on a command like bash
- You find yourself in a situation where all users running bash will be in a shell
- Because setuid provides a dangerous pathway to privilege escalation, some container scanners will report the presence of files with the setuid bit set
- Setuid was introduced in a time where we did not need high granularity over roles: either you had root privileges or you did not

Linux Capabilities

- To provide more granularity over privileges, version 2.2 of the Linux kernel introduced *capabilities*
- There are over 30 different capabilities which can be assigned to threads in order to determine whether that thread can perform certain actions
- You can see the capabilities via the getpcaps command with input the ID of the process (get id with command ps)

Linux Capabilities

```
vagrant@vagrant:~$ ps
  PID TTY          TIME CMD
22355 pts/0    00:00:00 bash
25058 pts/0    00:00:00 ps
vagrant@vagrant:~$ getpcaps 22355
Capabilities for '22355': =
```

non-root

```
vagrant@vagrant:~$ sudo bash
root@vagrant:~# ps
  PID TTY          TIME CMD
25061 pts/0    00:00:00 sudo
25062 pts/0    00:00:00 bash
25070 pts/0    00:00:00 ps
root@vagrant:~# getpcaps 25062
Capabilities for '25062': = cap_chown,cap_dac_override,cap_dac_read_search,
cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap
cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,
cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,
cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,
cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,
cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override
cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+ep
```

root

Assign Capabilities


```
vagrant@vagrant:~$ cp /bin/ping ./myping
vagrant@vagrant:~$ ls -l myping
-rwxr-xr-x 1 vagrant vagrant 64424 Feb 12 18:18 myping
vagrant@vagrant:~$ ./myping 10.0.0.1
ping: socket: Operation not permitted
```

```
vagrant@vagrant:~$ sudo setcap 'cap_net_raw+p' ./myping
vagrant@vagrant:~$ getcap ./myping
./myping = cap_net_raw+p
vagrant@vagrant:~$ ./myping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
```

Control Groups

- Control groups are a fundamental building block for building containers
- **Cgroups** limit the resources that a group of processes can use
- From a security perspective, they ensure that one process can not affects the behavior of other processes by hogging all the resources
- Control groups are organized in *hierarchies*, and in particular a hierarchy for each type of resource being managed
- Each hierarchy is managed by a cgroup controller
- Linux processes inherit the cgroup of its parent

Control Groups

- The Linux kernel communicates information regarding cgroups through a set of pseudo-filesystems typically at /sys/fs/cgroup
- We can see them by listing the content of that directory
- Managing cgroups involves reading and writing to files and directories within these hierarchies

```
root@vagrant:/sys/fs/cgroup$ ls
blkio    cpu,cpuacct  freezer  net_cls           perf_event  systemd
cpu      cpuset       hugetlb  net_cls,net_prio  pids        unified
cpuacct  devices      memory   net_prio          rdma
```

Creating cgroups

- Creating a subdirectory in the memory directory creates a cgroup
- The kernel automatically populates the directory with the various files that represent parameters and statistics about the cgroup
- When you start a container, the runtime creates cgroups for it
- By default resources (e.g., memory) are not limited
- If a process is allowed to consume unlimited memory, it can starve other processes on the same host
- **Resource exhaustion attack:** use as much memory as possible

Docker with cgroups

- Docker automatically creates its own cgropus for each type
- We can see them by looking at directories called docker within the cgroups hierarchy
- When you start a new container, it automatically creates another set of cgroups within the docker cgroups

Linux Namespaces

- If cgroups control the resources a process can use, namespaces control what it can see
- By adding a process to a namespace, you limit the resources that are visible to that process
- A process is always in exactly one namespace of each type
- At initialization, a Linux system has a single namespace of each type, but you can create additional ones and assign processes to them
- Use lsns to see namespaces
- Let's see how to use namespaces to create something behaving like a container

Isolating the hostname

- We start from the Unix Timesharing System, which covers the domain names and hostname
- By adding a process to its own UTS namespace, you can change the hostname for this process independently of the hostname of the machine or VM it is running incd
- Open a terminal in linux and check *hostname*
- For containers, ID are randomly assigned and used as hostnames
- The container has its own UTS namespace, so its own hostname

Unshare

- To obtain an effect similar to that obtained with container, we can use the unshare command
- When you run a program the kernel creates a new process and executes the program in it
- Objective: run a command with some namespace unshared from the parent
- Need root privileges to do this
- This is a key component on how containers work: namespaces give them a set of resources that are independent of the host machine and of other containers

Isolate Process IDs

- By running the ps command inside a Docker container you see only the processes running inside the container, not that of the host
- This is achieved thanks to the process ID namespace, which restricts the set of process IDs that are visible
- We can use again unshare and specify that we want a new PID namespace
- We see errors in the format *command: process ID: message*
- We need to fork
- If you run ps inside the fork, we see all the processes in the whole host: not what you would expect from a container

Isolate Process IDs

- Irrespective of the process ID namespace it is running in, ps looks into /proc for information about running processes
- If we want ps to return only the information about the processes inside the new namespace, we need a separate copy of /proc
- Given that /proc is directly under root, this means changing the root directory

Building Images

- Most of the times, when using Docker, we resort to the docker build command
- This follows the instructions from a file called Dockerfile to create an image
- However, Docker build should be carefully managed from a security point of view
- When you run a docker command, the command line tool creates an API request that it sends to the Docker daemon via the Docker socket
- Any software having access to the Docker socket can send API requests to the daemon

Building Images

- The Docker daemon is along running process that does the work of running and managing both container and container images
- In order to create a container, we need root privileges, as we need to crete namespaces
- Imagine you want to dedicate a machine to build container images and store them in a registry
- Using Docker, your machine needs to run the daemon, which has far more capabilities beyond building and interacting with registries
- **Any user who can trigger a docker build can perform a docker run**

Image Layers

- The vast majority of container image builds are defined through a Dockerfile
- Dockerfile gives a series of instructions, each of which results in either a filesystem layer or a change to the image configuration
- Anyone who has access to a container image can access any file included in that image
- From a security perspective, you want to avoid including sensitive information such as passwords or tokens in an image

Image Layers

- The fact that every layer is stored separately means that you have to be careful not to store sensitive data, even if a subsequent layer removes it

```
FROM alpine
RUN echo "top-secret" > /password.txt
RUN rm /password.txt
```

- Seems like one layer creates a file and the next one deletes it
- However, the sensitive data is still included in the image

Image Layers

SPRITZ Security & Privacy Research Group

Università degli Studi di Padova

```
vagrant@vagrant:~$ docker save sensitive > sensitive.tar
vagrant@vagrant:~$ mkdir sensitive
vagrant@vagrant:~$ cd sensitive
vagrant@vagrant:~$ tar -xf ../sensitive.tar
vagrant@vagrant:~/sensitive$ ls
0c247e34f78415b03155dae3d2ec7ed941801aa8aeb3cb4301eab9519302a3b9.json
552e9f7172fe87f322d421aec2b124691cd80edc9ba3fef842b0564e7a86041e
818c5ec07b8ee1d0d3ed6e12875d9d597c210b488e74667a03a58cd43dc9be1a
8e635d6264340a45901f63d2a18ea5bc8c680919e07191e4ef276860952d0399
manifest.json
```

- The config includes the history of the commands that were run to construct this container

Image Layers

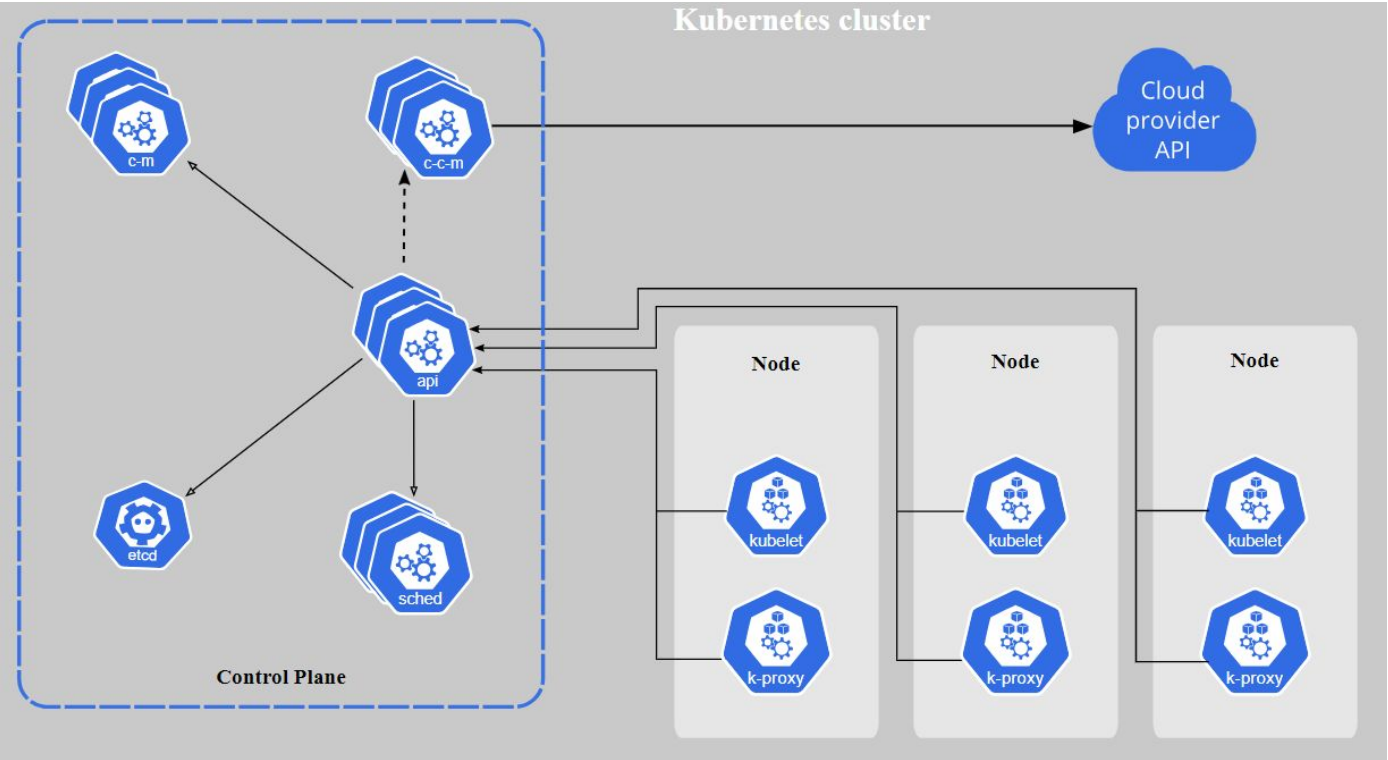
SPRITZ Security & Privacy Research Group
Università degli Studi di Padova

```
vagrant@vagrant:~/sensitive$ cat 0c247*.json | jq '.history'
[
  {
    "created": "2019-10-21T17:21:42.078618181Z",
    "created_by": "/bin/sh -c #(nop) ADD
    file:fe1f09249227e2da2089afb4d07e16cbf832eeb804120074acd2b8192876cd28 in / "
  },
  {
    "created": "2019-10-21T17:21:42.387111039Z",
    "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
    "empty_layer": true
  },
  {
    "created": "2019-12-16T13:50:43.914972168Z",
    "created_by": "/bin/sh -c echo \"top-secret\" > /password.txt"
  },
  {
    "created": "2019-12-16T13:50:45.085349285Z",
    "created_by": "/bin/sh -c rm /password.txt"
  }
]
```

Kubernetes (K8s)

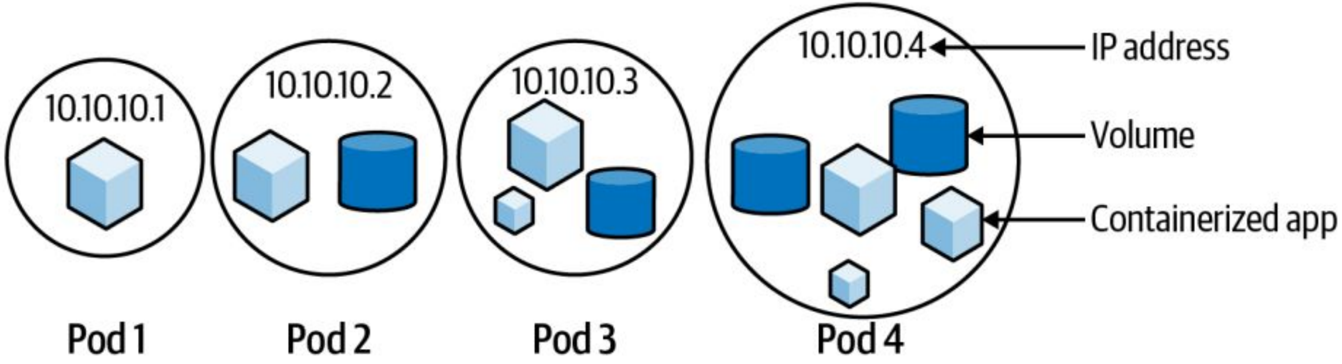
- Is an open-source container orchestration system for automating software deployment, scaling, and managed
- Originally designed by Google, now maintained by the Cloud Native Computing Foundation
- It provides a framework to run distributed system resiliently, taking care of scaling and failover
- It operates at the container level, and when you deploy it you get a **cluster**

Kubernetes (K8s)



Pods and their Content

- A Pod is the smallest deployable unit you can ask Kubernetes to run
- It is an environment where multiple containers can run, and defines a trust boundary encompassing all the containers in it (including identity and access)
- It has its own IP address, can mount volumes, and its namespaces surround the containers created by the container runtime



Kubelet

- The lifecycle of a pod is controlled by the **kubelet**, the Kubernetes API server's deputy, deployed on each node in the cluster to manage and run containers
- The kubelet attaches pods to a Container Network Interface (CNI), whose traffic is treated as layer 4 TCP/IP
- If traffic is unencrypted it may be sniffed by a compromised pod or node

Defaults

- Flat topology: every pod can see and talk to every other pod in the cluster
- No securityContext: workloads can escalate to host network interface controller
- No environmental restrictions: workloads can query their host and cloud metadata
- No identity for workloads
- No encryption on the wire
- However, depending on the communications we can have difference namespaces that provide limited views to pods in a node

Conclusions

- Containers and orchestrators are fundamental components of modern software production systems
- However, they may come with unsecure configurations, poisoned, or reuse pieces of code that are not secure
- Defending against supply chain attacks is a top-priority for companies