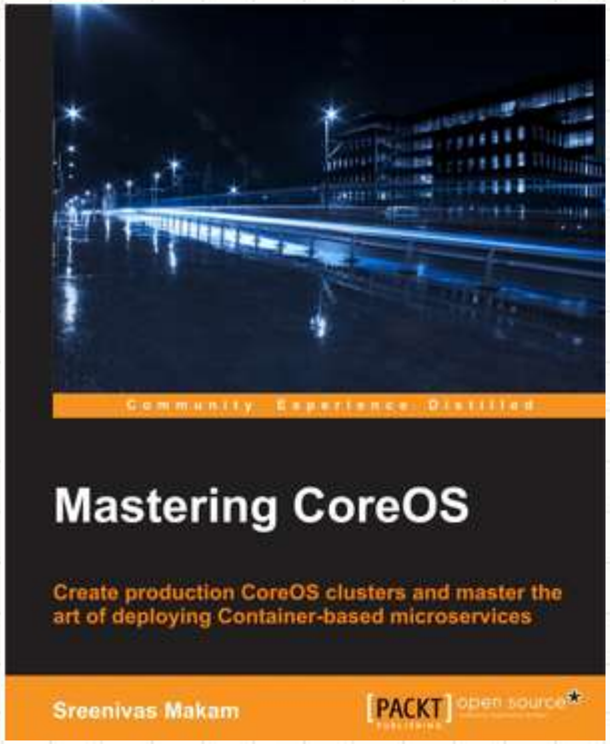
Docker Security Overview (As of release 1.12)

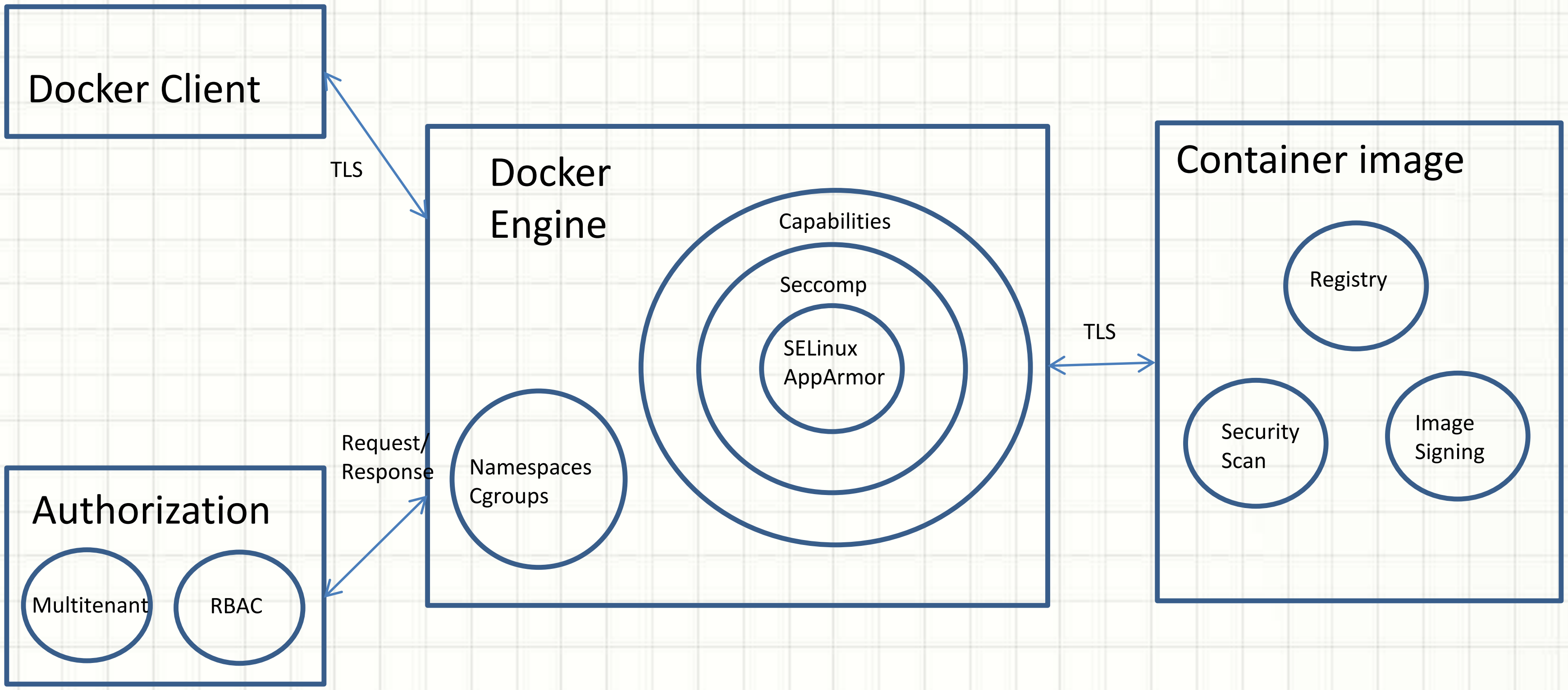
Presenter Name: Sreenivas Makam
Presented at: Docker Meetup Bangalore
Presentation Date: July 9, 2016

About me

- Senior Engineering Manager at Cisco Systems Data Center group
- Author of “Mastering CoreOS” https://www.packtpub.com/networking-and-servers/mastering-coreos/)
- Docker Captain(https://www.docker.com/community/docker-captains)
- Blog: https://sreeninet.wordpress.com/
- Code: https://github.com/smakam
- Linkedin: https://in.linkedin.com/in/sreenivasmakam
- Twitter: @srmakam


Community Experience Distilled
Mastering CoreOS
Create production CoreOS clusters and master the art of deploying Container-based microservices
Sreenivas Makam
[PACKT] open source

Docker Security modules



Linux kernel - Container Security support

- Namespaces – PID, Mount, Network, IPC, UTC, User.
- Cgroups – Limit CPU, Memory, IO
- Capabilities – Reduced root access. 36 capabilities to control as of kernel 3.19.0.21.
- Seccomp profiles – Control kernel system calls. 300+ system calls available that can be controlled using these profiles.
- Special kernel modules like AppArmor, SELinux – Provides granular control over Kernel resources.

Namespaces – PID, Mount

- With PID namespace, each Container gets its own process ID namespace.
- With Mount namespace, each Container gets its own copy of filesystem.

```
docker run -ti --name ubuntu1 -v /usr:/ubuntu1 ubuntu bash
docker run -ti --name ubuntu2 -v /usr:/ubuntu2 ubuntu bash
```

PID namespace:

Ubuntu1 Container:

```
root@3a1bf12161c9:/# ps
PID TTY TIME CMD
1 ? 00:00:00 bash
15 ? 00:00:00 ps
```

Ubuntu2 Container:

```
root@8beb85abe6a5:/# ps
PID TTY TIME CMD
1 ? 00:00:00 bash
14 ? 00:00:00 ps
```

Host:

```
$ ps -eaf|grep root | grep bash
root 5413 1697 0 05:54 pts/28 00:00:00 bash
root 5516 1697 0 05:54 pts/31 00:00:00 bash
```

Mount namespace:

Ubuntu1 Container:

```
root@3a1bf12161c9:/# ls /
bin dev home lib64 mnt proc run srv tmp usr
boot etc lib media opt root sbin sys ubuntu1 var
```

Ubuntu2 Container:

```
root@8beb85abe6a5:/# ls /
bin dev home lib64 mnt proc run srv tmp usr
boot etc lib media opt root sbin sys ubuntu2 var
```

Namespaces – Network, UTS

- With Network namespace, each Container gets its own interfaces along with IP address.

Ubuntu1 Container:

```
root@3a1bf12161c9:/# ifconfig
eth0 Link encap:Ethernet HWaddr
02:42:ac:15:00:02 inet addr:172.21.0.2
Bcast:0.0.0.0 Mask:255.255.0.0 inet6 addr:
fe80::42:acff:fe15:2/64
```

Ubuntu2 Container:

```
root@8beb85abe6a5:/# ifconfig
eth0 Link encap:Ethernet HWaddr
02:42:ac:15:00:03 inet addr:172.21.0.3
Bcast:0.0.0.0 Mask:255.255.0.0 inet6 addr:
fe80::42:acff:fe15:3/64
```

- With UTS namespace, each Container gets its own hostname and domainname.

Ubuntu1 Container:

```
root@3a1bf12161c9:/# hostname
3a1bf12161c9
```

Ubuntu2 Container:

```
root@8beb85abe6a5:/# hostname
8beb85abe6a5
```

Namespaces - IPC

- IPC namespace isolates Message queues, Semaphores, Shared memory

Ubuntu 1 Container:
Create shared memory:

```
root@3a1bf12161c9:/# ipcmk -M 100
Shared memory id: 0
```

Display shared memory:

```
root@3a1bf12161c9:/# ipcs -m
------ Shared Memory Segments --------
key shmid owner perms bytes nattch
status
0x2fba9021 0 root 644 100 0
```

Ubuntu 2 Container:
Create shared memory:

```
root@8beb85abe6a5:/# ipcmk -M 100
Shared memory id: 0
```

Display shared memory:

```
root@8beb85abe6a5:/# ipcs -m
------ Shared Memory Segments --------
key shmid owner perms bytes nattch
status
0x1f91e62c 0 root 644 100 0
```

Namespaces - User

- With User namespace, userid and groupid in a namespace is different from host machine’s userid and groupid for the same user and group.
- User namespaces are available from Linux kernel versions > 3.8.
- For example, root user inside Container is not root inside host machine. This provides greater security.
- Docker introduced support for user namespace in version 1.10.
- To use user namespace, Docker daemon needs to be started with “–userns-remap=username/uid:groupname/gid”. Using “default” for username will create “dockremap” user.

Ubuntu1 Container:

```
root@3a1bf12161c9:/# id
uid=0(root) gid=0(root) groups=0(root)
```

Host:

```
$ ps -eaf|grep bash
231072    4080  4040  0 22:45 pts/13
00:00:00 bash

$ cat /proc/4080/uid_map
0     231072      65536
```

(userid 0 in Container is mapped to 231072 in host)

cgroups

- cgroups is a Linux kernel feature that provides capability to restrict resources like cpu, memory, io, network bandwidth among a set of processes.
- Docker allows to create Containers using cgroup feature which allows for resource control for the specific Container.
- Following is a Container created with user space memory limited to 500m, kernel memory limited to 50m, cpu share to 512, blkioweight to 400. (cpu share and blkioweight are ratios)

```
docker run -it -m 500M --kernel-memory 50M --cpu-shares 512 --blkio-weight 400 --
name ubuntu1 ubuntu bash
```

Capabilities

- In Linux, root user typically has all privileges enabled. Capabilities allow finer control for the capabilities that can be allowed for root user.
- In Linux 3.19.0.21, there are 36 capabilities. All capabilities can be seen in “/usr/include/linux/capability.h”.
- Examples of capabilities are setuid, setgid, socket access, set time.
- Docker turns on only 14 capabilities by default for Containers started with default options.
- Capabilities like insert/remove kernel modules, system clock manipulation are blocked.

Container with raw net capability turned off:

```
docker run -ti --name ubuntu1 --cap-drop=net_raw ubuntu bash
```

Network access blocked:

```
# ping google.com
ping: icmp open socket: Operation not permitted
```

Seccomp

- Linux kernel feature that limits the system calls that a process can make based on the specified profile.
- Docker uses Seccomp to control the system calls that Container can make.
- Examples of system calls – bind, accept, fork.
- Docker disables around 44 of 300+ system calls for Containers started with default options. Example of disabled system calls include mount, settimeofday.
- Default Docker Seccomp profile is available here.

Profile disabling chmod system call:

```
{ "defaultAction":
"SCMP_ACT_ALLOW",
"syscalls": [ { "name": "chmod",
"action": "SCMP_ACT_ERRNO" } ] }
```

Seccomp illustration:

```
$ docker run --rm -it --security-opt
seccomp:/home/smakam14/seccomp/profil
e.json busybox chmod 400 /etc/hosts
chmod: /etc/hosts: Operation not
permitted
```

Linux kernel Security modules – AppArmor, SELinux

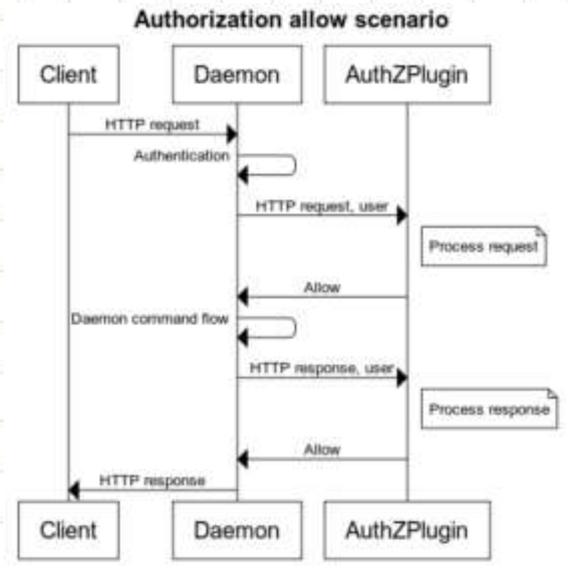
- Both AppArmor and SELinux are kernel modules that gives fine grained control to restrict access to system resources.
- When AppArmor is active for an application, the operating system allows the application to access only those files and folders that are mentioned in its security profile.
- SElinux is a labeling system. Every process, file, directory, network ports, devices has a label assigned to it. We write rules to control the access of a process label to an a object label like a file. The kernel enforces the rules specified in the policy.
- In Redhat distributions, SELinux is supported. Ubuntu distributions supports AppArmor.
- AppArmor profiles are easy to create, SELinux is difficult. SELinux profiles are more comprehensive compared to AppArmor.

Docker Engine Secure access

- Docker engine runs as a daemon and by default listens on the Unix socket, “unix:///var/run/docker.sock”.
- For accessing Docker engine remotely, “http” or “https” using TLS can be used.
- “http” is not advised to be used for security reasons.
- “https” with TLS provides confidentiality, authentication as well as integrity.
- Certificates can be used to establish identity of client and server.
- For testing purposes, Certificates can be generated using Openssl. For commercial purposes, certificates can be purchased from sources like CA.

Authorization plugin

- Authorization plugin can provide granular access to the Docker daemon based on userid, groupid, command executed, command arguments, time of day etc.
- Authorization plugin informs Docker daemon if the specific command can be allowed or not based on the policy and the command executed.
- Twistlock authz broker is one of the first plugins that is currently available.
- In Twistlock case, policy is created as a JSON file and is given as argument to Docker daemon.
- User identity support is not yet available in Docker engine.
- Docker Data Center and Docker cloud provides RBAC and Multi-tenant support.



Policy.json:
{"name":"policy_1","users":[""],"actions":["container"] , "readonly":true} – Read-only Container policy

Starting Docker daemon with auth policy.json:
docker run -d --restart=always -v /var/lib/authz-broker/policy.json:/var/lib/authz-broker/policy.json -v /run/docker/plugins/:/run/docker/plugins twistlock/authz-broker

Container image signing

- Container images needs to be signed so that the client knows that image is coming from a trusted source and that it is not tampered with.
- Content publisher takes care of signing Container image and pushing it into the registry.
- The Docker content trust is an implementation of the Notary open source project. The Notary open source project is based on The Update Framework (TUF) project.
- When content trust is enabled, we can pull only signed images.
- When content trust is not enabled, both signed and unsigned images can be pulled.
- Docker content trust is enabled with “export DOCKER_CONTENT_TRUST=1”.
- When the publisher pushes the image for the first time using docker push, there is a need to enter a passphrase for the root key .
- When pushing new image or new image version, publisher needs to enter passphrase for repository key.
- Docker has also added support for hardware keys using Yubikey
- Keys should be saved safely and backups should be taken.

Container Image scanning

- Docker Security Scan scans Container images and reports vulnerabilities.
- Scanning is done by comparing each Container layer component with CVE databases.
- Additional binary scan is done to make sure that the package is not tampered with.
- Pro-active notification is given to both the publisher and user of Container images.
- Available currently in Docker hub and Docker cloud. Will be added soon to Docker data center.

Docker Security - Best Practices

Docker enables Security by default and the default options suffice most of the needs. Following are few best practices:

- Have separate containers for each micro-service keeping Container image size small.
- Don’t put ssh inside container, “docker exec” can be used to ssh to Container.
- Use only signed Container images.
- Mount devices and volumes as read-only.
- Run application as non-root. If root access is needed, run as root only for limited operations using features like Capabilities, Seccomp, SELinux/AppArmor.
- Keep OS secure with regular updates. Using Container optimized OS is an option here since they provide automatic pushed update.
- Store root keys, passphrase in a safe place and not expose in Dockerfile. Docker has plans to manage keys with Docker datacenter.
- Use Docker official images. These images are curated by Docker so that the highest quality and security is maintained for the official images.
- Use Container security scanning to check for vulnerabilities.
- Use TLS for remote Docker daemon access.

References

- Docker Security documentation(https://docs.docker.com/engine/security/security/)
- Container namespaces (http://crosbymichael.com/creating-containers-part-1.html)
- Docker Security article series (https://opensource.com/business/15/3/docker-security-tuning)
- Docker Security blog series (https://sreeninet.wordpress.com/2016/03/06/docker-security-part-1overview/)
- Docker Security scan(https://blog.docker.com/2016/05/docker-security-scanning/)
- Authorization plugin (https://www.twistlock.com/2016/02/18/docker-authz-plugins-twistlocks-contribution-to-the-docker-community/)