OWASP Docker Top 10

The Then Most Important Aspects To Build a Secure Containerized Environment.

https://owasp.org

Table of Contents

Introduction

Implementing a containerized environment is changing a lot not only how deployments are done but it also has a huge impact on a system and networking level and how hardware and network resources are being used.

This document is helping you to secure your containerized environment and keep it secure.

Application Security?

There are often misunderstandings of what the security impacts - negative or positive - are supposed to be when using Docker.

Docker as any other containerization technology doesn't solve application security problems. It doesn't help doing input validation and it doesn't provide protecting against SQL injection. For application security risks OWASP provides a lot of other useful documents, starting from the OWASP Top 10 over the OWASP Proactive Controls to the OWASP Application Security Verification standard -- just to name a few.

Container Security is mostly about system and network security and a secure architectural design.

This indicates that it is best before you start using containerization, you should plan your environment in a secure manner. Some points are just difficult or costly to change later when you started already rolling out your containerization in production.

Shift in Paradigm: new vectors

Looking at it from the perspective of the classical world, especially in system and network areas containerization means big changes to your environment. Those changes are opening up new potential attack surfaces. Special care has to be taken so that no network and system security problems arise.

Apart from these technical areas there are two non-technical points:

- Docker with its 5 years is a relatively new technology. Subtracting the time for maturing and adoption the time span is even shorter. Every new technology needs time until the knowledge of the technology and their best practices becomes common knowledge.
- While container solutions might offer benefits for the developer, the technology is not simple from the security perspective. Not being simple is what makes security more difficult, a.k.a. the KISS principle -- keep it simple and stupid.

This is what this document is trying to help you with: It provides you with the knowledge to avoid common pitfalls in the system and network area and it tries to get a handle on the complexity.

Document Structure

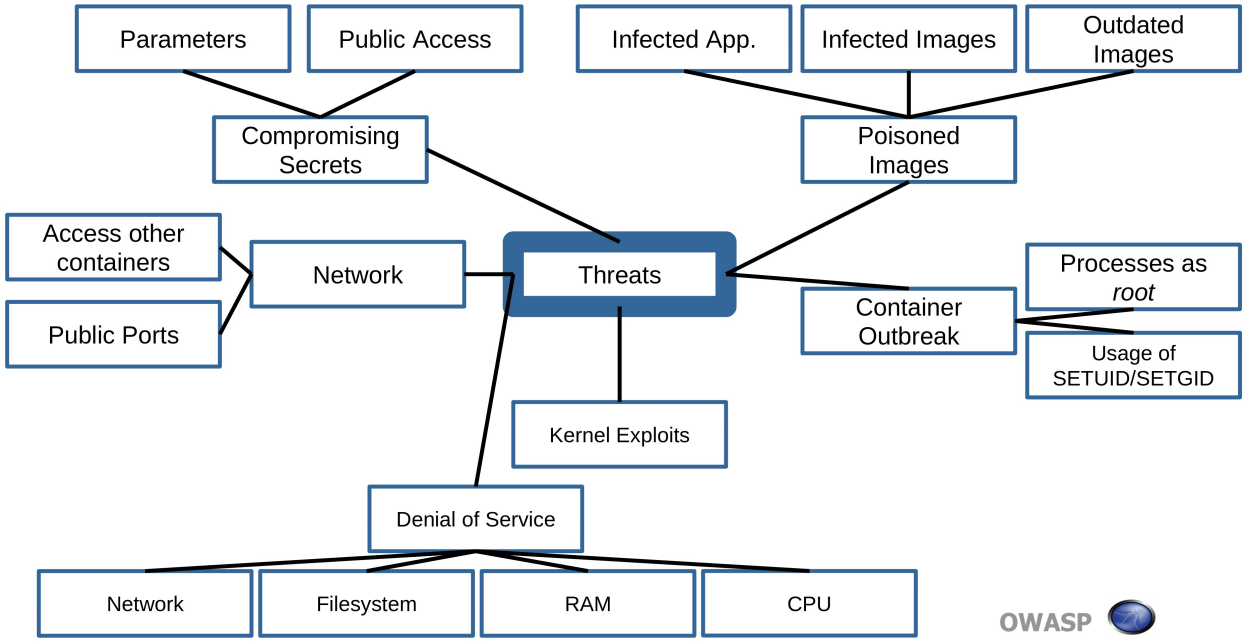
In order to achieve this, this document first does an analysis of the threats caused by the technology. This is the basis for the ten points to follow.

Threat Modeling

The classical approach how to secure an environment is looking at it from the attacker perspective and enumerate vectors for an attack. This is what this chapter is about.

Those vectors will help you to define what needs to be protected. With this definition one can put security controls in place to provide a baseline protection and beyond. This is what the ten controls in the following chapters are about.

The following image gives an overview of threats in docker.



Threat 1: Container Escape (System)

In this scenario the application is insecure in a way that some kind of shell access is possible. So the attacker managed e.g. from the internet to successfully stage an attack in which he has managed to escape the application and ended up to be in the container. The container is as the name indicates supposed to contain him.

In a second stage he would try to escape the container, either as the container user from a view of the host or with a kernel exploit. In the first scenario he would just end up with user privileges on the host. In the second scenario he would be root on the host system which gives him control over all containers running on that host.

Threat 2: Other Containers via Network

This scenario has the same first stage as the previous one. The attacker has also shell access but then he chooses to attack another container through the network. That could either be from the same application, a different application from the same customer, or in a multi-tenant environment one from another customer.

Threat 3: Attacking the Orchestration Tool via Network

This scenario has the same first vector as the previous two. The attacker has shell access within the container but he chooses to attack the management interfaces or other attacks surfaces of the orchestration tool -- the management back-plane. In 2018 almost every tool has had a weakness here

which was a default open management interface. "Open" means in the worst case an open port without authentication. (citations needed).

Threat 4: Attacking the Host via Network

This again has the same first vector as the one mentioned before. With his shell access he attacks an open port from the host. If it is weakly protected or not at all he get's user - or worse - root access to the host.

Threat 5: Attacking other Resources via Network

This is basically a threat which collects all remaining network-based threats into one bucket.

This again has the same first vector as the one mentioned before. With his shell access he finds e.g. an unprotected network-based file system which is shared among the containers where he could read or even modify data. Another possibility would be resources like an Active or LDAP Directory. Yet another resource could be e.g. a Jenkins which has somebody configured too open and is accessible from the container.

(clarification needed because of ARP spoofing & switch): Also it could be possible that the attacker installs a network sniffer into the container he hijacked so that he might be able to read traffic from other containers

Threat 6: Resource Starvation

The underlying vector is due to a security condition from another container running on the same host. The security condition could be either to the fact that the other container is eating up resources which could be CPU cycles, RAM, network or disk-I/O.

It could also be that the container has a host file system mounted which the attacker has been filling up which causes problem on the host which in turn affects other containers.

Threat 7: Host compromise

Whereas the previous threat the attacker managed indirectly over the host to affect another / other containers, here the attacker has compromised the host -- either through another container or through the network.

Threat 8: Integrity of Images

The CD pipeline could involve several hops where the mini operating system image is been passed from one step to the next until it reaches the deployment.

Every hop is a potential attack surface for the attacker. If an attacker manages to get foot into one step and there's no integrity check whether what will be deployed is what should be deployed there is the threat that on behalf of the attacker images with his malicious payloads are being deployed.

Overview OWASP Docker Top 10

Title	Description
D01 - Secure User Mapping	Most often the application within the container runs with the default administrative privileges: root. This violates the least privilege principle and gives an attacker better chances further extending his activities if he manages to break out of the application into the container. From the host perspective the application should never run as root.
D02 - Patch Management Strategy	The host, the containment technology, the orchestration solution and the minimal operating system images in the container will have security bugs. Once publicly known it is vital for your security posture to address those bugs in a timely fashion. For all those components mentioned you need to decide when you apply regular and emergency patches before you put those into production.
D03 - Network Segmentation and Firewalling	You properly need to design your network upfront. Management interfaces from the orchestration tool and especially network services from the host are crucial and need to be protected on a network level. Also make sure that all other network based microservices are only exposed to the legitimate consumer of this microservice and not to the whole network.
D04 - Secure Defaults and Hardening	Depending on your choice of host and container operating system and orchestration tool you have to take care that no unneeded components are installed or started. Also all needed components need to be properly configured and locked down.
D05 - Maintain Security Contexts	Mixing production containers on one host with other stages of undefined or less secure containers may open a backdoor to your production. Also mixing e.g. frontend with backend services on one host may have negative security impacts.
D06 - Protect Secrets	Authentication and authorization of a microservice against a peer or a third party requires secrets to be provided. For an attacker those secrets potentially enables him to access more of your data or services. Thus any passwords, tokens, private keys or certificates need to be protected as good as possible.
D07 - Resource Protection	As all containers share the same physical CPU, disks, memory and networks. Those physical resources need to be protected so that a single container running out of control -- deliberately or not -- doesn't affect any other container's resources.
D08 - Container Image Integrity and Origin	The minimal operating system in the container runs your code and needs to be trustworthy, starting from the origin up until the deployment. You need to make sure that all transfers and images at rest haven't been tampered with.
D09 - Follow Immutable Paradigm	Often container images don't need to write into their filesystem or a mounted filesystem, once set up and deployed. In those cases you have a extra security benefit if you start the containers in read-only mode.
D10 - Logging	For your container image, orchestration tool and host you need to log all security relevant events on a system and API level. All logs should be remote, they should contain a common timestamp and they should be tamper proof. Your application should also provide remote logging.

D01 - Secure User Mapping

Threat Scenarios

The threat is here that a microservice is being offered to run under `root` in the container. If the service contains a weakness the attacker has full privileges within the container. While there's still some default protection left (Linux capabilities, either AppArmor or SELinux profiles) it removes one layer of protection. This extra layer broadens the attack surface. It also violates the least privilege principle [1] and from the OWASP perspective an insecure default.

For privileged containers (`--privileged`) a breakout from the microservice into the container is almost comparable to run without any container. Privileged containers endanger your whole host and all other containers.

How Do I prevent?

It is important to run your microservice with the least privilege possible.

First of all: Never use the `--privileged` flag. It gives all so-called capabilities (see D04) to the container and it can access host devices (`/dev`) including disks, and also has access to the `/sys` and `/proc` filesystem. And with a little work the container can even load kernel modules on the host [2]. The good thing is that containers are per default unprivileged. You would have to configure them explicitly to run privileged.

However still running your microservice under a different user as root requires configuration. You need to configure your mini distribution of your container to both contain a user (and maybe a group) and your service needs to make use of this user and group.

Basically there are two choices.

In a simple container scenario if you build your container you have to add `RUN useradd <username>` or `RUN adduser <username>` with the appropriate parameters -- respectively the same applies for group IDs. Then, before you start the microservice, the `USER <username>` [3] switches to this user. Please note that a standard web server wants to use a port like 80 or 443. Configuring a user doesn't let you bind the server on any port below 1024. There's no need at all to bind to a low port for any service. You need to configure a higher port and map this port accordingly with the expose command [4]. Your mileage may vary if you're using an orchestration tool.

The second choice would be using Linux user namespaces. Namespaces are a general means to provide to a container a different (faked) view of Linux kernel resources. There are different resources available like User, Network, PID, IPC, see `namespaces(7)` . In the case of user namespaces a container could be provided with a his view of a standard root user whereas the host kernel maps this to a different user ID. More, see [5], `cgroup_namespaces(7)` and `user_namespaces(7)` .

The catch using namespaces is that you can only run one namespace at a time. If you run user namespacing you e.g. can't use network namespacing on the same host [6]. Also, all your containers on a host will be defaulted to it, unless you explicitly configure this differently per container.

In any case use user IDs which haven't been taken yet. If you e.g. run a service in a container which maps outside the container to a `systemd` user, this is not necessarily better.

How can I find out?

Configuration

Depending on how you start your containers the first place is to have a look into the configuration / build file of your container whether it contains a user.

Runtime

Have a look in the process list of the host, or use `docker top` or `docker inspect`.

1) `ps auxwf`

2) `docker top <containerID>` or `for d in $(docker ps -q); do docker top $d; done`

3) Determine the value of the key `Config/User` in `docker inspect <containerID>`. For all running containers: `docker inspect $(docker ps -q) --format='{{.Config.User}}'`

User namespaces

The files `/etc/subuid` and `/etc/subgid` do the UID mapping for all containers. If they don't exist and `/var/lib/docker/` doesn't contain any other entries owned by `root:root` you're not using any UID remapping. On the other hand if those files exist and there are files in that directory you still need to check whether your docker daemon was started with `--userns-remap` or the config file `/etc/docker/daemon.json` was used.

References

- [1] OWASP: Security by Design Principles
- [3] Docker Docs: USER command
- [4] Docker Docs: EXPOSE command
- [5] Docker Docs: Isolate containers with a user namespace
- [6] Docker Docs: User namespace known limitations

Commercial

- [2] How I Hacked Play-with-Docker and Remotely Ran Code on the Host

D02 - Patch Management Strategy

Not patching the infrastructure in a timely fashion is still the most frequent security problem in the IT industry which proved Viruses like WannaCry or NotPetya. Most software defects from "off the shelf software" are well known for some time before exploits are written and used. Sometimes not even a sophisticated exploit is needed. All you need to do is patching those known vulnerabilities soon enough.

This is similar to OWASP Top 10's "Known Vulnerabilities" [1].

It's necessary to agree and have a policy or at least a common understanding when certain patches will be applied. Often this strategy would be a task of an Information Security Officer, an ISO or CISO. Not having an (C)ISO is no excuse for not having a patch management strategy. Please note that the term patch management strategy in this paragraph the scope is not primarily a technical one. Synonymous terms for patch management strategy is patch management policy or plan, or security SLA.

Threat Scenarios

The worst thing which can happen to your container environment is that either the host(s) are compromised or the orchestration tool. The first would enable the attacker to control all the containers running on the host, the second will enable the attacker to control all containers on all hosts which the software is managing.

The most severe threats to the host are kernel exploits from within a container through abuse of Linux sys(tem)calls which lead to root access [2]. Also the orchestration software has interfaces whose defaults were weak and have shown numerous problems in the past [3],[4].

While threats from the Linux kernel can be partly mitigated by constraining syscalls further (see D4) and network access restrictions (D3) can be applied to reduce the network vector for the orchestration, it is important to keep in mind that you can't mitigate future security problems, like from remaining syscalls, other than by patching. The likelihood of such an incident might be small, however the impact is huge, thus resulting in a high risk.

Patching timely makes sure that your software you are using for your infrastructure is always secure and you avoid known vulnerabilities.

Another threat arises from any Linux services on the host. Also if the host configuration is reasonable secured (see D3 and D4) e.g. a vulnerable `sshd` poses a threat to your host too. If the services is not secured via network and configuration, the risk is higher.

You should also keep an eye on the support life time of each "ops" component used. If e.g. the host OS or orchestration software has run out of support, you likely won't be able to address security issues.

How Do I prevent?

Different Patch Domains

Maintaining your infrastructure software is not as straightforward though as there are four different "patch domains":

- Images: the container distribution
- Container software: Docker
- Orchestration software (Kubernetes, Mesos, OpenShift, ...)
- Host: operating system

While the first domain of patching seems easy at the first glance updating the Container software is not seldom postponed. Same applies for the orchestration tool and the host as they are core components.

Have a migration plan for EOL support for each domain mentioned.

When to patch what?

In short: patch often and if possible automated.

Depending on the patch domains mentioned above there are different approaches how proper patching can be achieved. Important is to have a patch strategy for each component. Your patch strategy should handle regular and emergency patches. You also need to be prepared for testing patches and rollback procedures.

If you aren't in an environment which has change or patch policies or processes, the following is recommended (test procedures omitted for simplicity):

- Define a time span in which pending patches will be applied on a regular basis. This process should be automated.
- This can be different for each patch domain -- as the risk might be different -- but it doesn't have to. It may differ due to different risks: An exposed container, an API from your orchestration software or a severe kernel bug a higher risk then container the DB backend or a piece of middleware.
- Execute patch cycles and monitor them for success and failures, see below.
- For critical patches to your environment where the time span between the regular patches is too large for an attacker you need to define a policy for emergency patches which need to be followed in such a case. You also need a team which tracks critical vulnerabilities and patches, e.g. through vendor announcements or through security mailing lists. Emergency patches are normally applied within days or about a week.

Keep in mind that some patches require a restart of their service, a new deployment (container image) or even a reboot (host) to become effective. If this won't be done, your patching otherwise could be as effective as just not to patch. Technical details when to restart a service, a host or initiate a new deployment need to be defined in the the patch plan too.

It helps a lot if you have planned for redundancy, so that e.g. a new deployment of a container or a reboot of a host won't affect your services.

How can I find out?

Depending on your role there are different approaches. If you are external or not involved you can just ask what the plan is for the different patch domains and have the plans explained. This can be supplemented by having a look at the systems. Also keep an eye on the continuity management.

Manual

Without doing deep researches you can gather good indicators on the host like

- `uptime`
- When were last patches applied (`rpm --qa --last` , `tail -f /var/log/dpkg.log`). Which patches are pending? (RHEL/CentOS: `echo n | yum check-update` , Suse/SLES: `zypper list-patches --severity important -g security` , Debian/Ubuntu: `echo n | apt-get upgrade`).
- `ls -lrt /boot/vmlinu*` vs. `uname -a`
- Have a look at the runtime of processes (`top` --> column `TIME+`) including e.g. `dockerd` , `docker-containerd*` and `kube*` processes
- Deleted files: `lsof +L1`

If your role is internal within the organization, you need to be sure that a patch management plan exists and is being properly executed. Depending where you start with your policy recommended is [5].

Automated

For the host: patch often! Every Linux vendor nowadays supports automated patching. For monitoring patches there are external tools available for authenticated vulnerability scans like OpenVAS [6]. But also all Linux operation systems provide builtin means notifying you for outstanding security patches.

The general idea for container images is though to deploy often and only freshly build containers. Scanning also here should never be used as a reactive measure but rather to verify that your that your patching works. For your container images there are a variety of solutions available [7]. Both use feed data on the CVEs available.

In any case it's also recommended to make use of plugins for your monitoring software notifying you of pending patches.

References

- [1] OWASP's Top 10 2017, A9: Using Components with Known Vulnerabilities
- [3]: Weak default of etcd in CoreOS 2.1: The security footgun in etcd
- [4]: cvedetails on Kubernetes, Rancher
- [5] OpenVAS.

Commercial

- [2] Blog of Aquasec: Dirty COW Vulnerability: Impact on Containers
- [5] TBD: ~~Good source (1) for patch management, light-weighted (not ITIL, nor ISO 2700x)~~
- [6] TBD: ~~what all needs to be listed here?~~

D04 - Secure Defaults and Hardening

While D03 - Network Segmentation and Firewalling aims for providing a layer of protection for any network based services on the host and the containers and the orchestration tool: it doesn't address the root cause. It mitigates often just the symptom. If there's a network service started which is not needed at all you should rather not start it in the first place. And if the service started is needed you should lock it down properly.

Threat Scenarios

Basically there are three "domains" where services can be attacked:

- Interfaces from the orchestration tool. Typically: dashboard, etcd, API
- Interfaces from the Host. Typically: RPC services, OpenSSHD, avahi, network based systemd-services
- Interfaces within the container, either from the microservice (e.g. spring-boot) or from the distribution.

How Do I prevent?

Orchestration / Host

For your orchestration tool it is crucial to know what service is running and whether it is protected properly or has a weak default configuration.

For your host the first step is picking the right distribution: A standard Ubuntu system is a standard Ubuntu system. There are distributions specialized on hosting containers, you should rather consider this. In any case install a minimal distribution -- think of a minimal bare metal system. And if you rather opted for a standard distribution: Desktop applications, compiler environments or any server applications have no business here. They all add an attack surface to a crucial system in your environment.

Support lifetime is another topic: When you pick an OS for the host, find out the EOL date.

In general you need to make sure you know what services are offered from each component in your LAN. Then you need to decide what do with each:

- Can it be stopped/disabled without affecting the operation?
- Can it be started only on the localhost interface or any other network interface?
- Is authentication configured for this service?
- Can a tcpwrapper (host) or any other config option narrow down the access to this service?
- Are there any known design flaws? Did you review the documentation in terms of security?

For services which cannot be turned off, reconfigured or hardened: This is where the network based protection (D03) should at least provide one layer of defense.

Also: If your host OS hiccups because of AppArmor or SELinux rules, never switch those additional protections off. Find the root causes in the system log file with the tools provided and relax those rule only.

Container

Also for the containers, best practice is: do no install unnecessary packages [1]. Alpine Linux has a smaller footprint and has per default less binaries on board. It still comes with a set of binaries like `wget` and `netcat` (provided by busybox) though. In a case of an application breakout into the container those binaries could help an attacker "phoning home" and retrieven some tool. If you want to put the bar higher you should look into "distroless" [2] images.

There are a couple of further options you should look into. What can affect the security of the host kernel are defective syscalls. In a worst case this can lead to a privilege escalation from a container as a user to root on the host. So-called capabilities are a superset from the syscalls.

Here are some defenses:

- Disable SUID/SGID bits (`--security-opt no-new-privileges`): even if you run as a user, SUID binaries could elevate privileges. Or use `--cap-drop=setuid --cap-drop=setgid` when applying the following.
- Drop more capabilities (`--cap-drop`): Docker restricts the so-called capabilities of a container from 38 (see `/usr/include/linux/capability.h`) to 14 (see `man 7 capabilities` and [3]). Likely you can drop a few like `net_bind_service` , `net_raw` and more, see [4]. `pscap` is your tool on the host to list capabilities. Never use `--cap-add=all` .
- If you need finer grained controls as the capabilities can provide you can control each of the >300 syscalls with seccomp with a profile in JSON (`--security-opt seccomp=mysecure.json`), see [5]. About 44 syscalls are already disabled by default. Do not use `unconfined` or `apparmor=unconfined` here.

Best practise is to settle which of the above you chose. Better do not mix capabilities setting and seccomp profiles.

How can I find out?

- Special attention is needed for your orchestration tool. There have been unprotected interfaces by (bad) design [6], [7]-[9].
- You can always scan the system from the same network to see what is exposed in this LAN. D03 describes how to do that.
- Better is to look onto the system.
  - Host: Log in with administrative privileges and see what's running using `netstat -tulpn | grep -v ESTABLISHED` or `lsof -i -Pn| grep -v ESTABLISHED` . This won't return the network sockets from the containers though.
  - In a container you can use those commands as well - if `netstat` or `lsof` is supplied by the image.
- Any services might also be protected by the host-based firewall. What rules will be applied via default varies from host OS to host OS. As just reading the output from `iptables -t nat -L -nv` and `iptables -L -nv` becomes in larger container environments quickly a tedious task. Thus here it's a good idea to also scan the LAN.

References

- [1] Docker's Best Practices
- [2] Google's FLOSS project distroless
- [3] Docker Documentation: Runtime privilege and Linux capabilities
- [5] Docker Documentation, Seccomp security profiles for Docker
- [6] Weak default of etcd in CoreOS 2.1: The security footgun in etcd
- [7] Kubernetes documentation: Controlling access to the Kubelet: Kubelets expose HTTPS endpoints which grant powerful control over the node and containers. By default Kubelets allow unauthenticated access to this API. Production clusters should enable Kubelet authentication and authorization.
- [8] Github: "Exploit" for the API in [7].
- [9] Medium: Analysis of a Kubernetes hack — Backdooring through kubelet. Incident because of an

open API, see [7].

Commercial

- [4] RedHat Blog: Secure your Container: One weird trick

D05 - Maintain Security Contexts

To have the investment into the powerful hardware pay off, it might sound appropriate to put as much containers directly on one single host.

From the security standpoint this is often questionable as different containers may have different security contexts and also different security statuses.

A backend container and a frontend container on the same host might be one concern as they have different information security values. A bigger issue is mixing production e.g. with a test or development environment. Production systems need to be available and development containers might contain code which isn't necessarily as secure. One shouldn't affect the other.

Most thoughts have to be put into multi tenant environments.

Threat Scenarios

- A student from the university has a part time job at a company. He just learned PHP programming and deploys his work into the CD chain of a company. The company has limited resources and bought only a very few of big iron hosts which serve all containers. The environment has rapidly and historically grown so that nobody had the resources to split production from test environment or the playground area. Unfortunately the student's application contains a remote execution vulnerability which internet scanning bots find this vulnerability and within a fingersnip exploit it. That means it broke out from the application he ended up in the container. Through this vulnerability the attacker goes shopping in the network and accessed either an insecured etcd or http(s) interface of the orchestration tool. Or he downloads an exploit as there's a similar vulnerability as Dirty COW [1] which grants him immediate root access to the big iron machine - including all containers.

This is a slightly exaggerated scenario. One can exchange the student from the university with an in-house developer who just did one mistake which is obvious looking at the application from the outside but was not visible to him.

One can also change the company to a different one providing a container service. And the developer to a client of the CaaS (Container as a Service) company. If one client of the CaaS provider does a similar mistake as the student, as a worst case scenario it could affect the whole CaaS environment's availability, confidentiality and integrity.

How Do I prevent?

Also if some of the threat scenarios might appear deliberately exaggerated, you should have gotten the picture:

As a general rule of thumb it's not recommended to mix containers with different security statuses or contexts.

- Put production containers on a separate host system and be careful who has he privilege deploying to this host. There should be no other containers allowed on this host.
- Looking at the information security value of your data you should also considering separating containers according to their contexts. Databases, middleware, authentication services, frontend and master components (cluster’s control plane of e.g. Kubernetes) shouldn't be on the same host.
- VMs (Virtual Machines) can be used in general to separate different security contexts from each other, like production and test. This is the minimum requirement when you are short in physical hardware and need run different tenants on one hardware.

How can I find out?

As an external auditor it's the best to get the system's architecture explained. In addition by logging in to the bare metal system you can check whether there are processes running which look like a VM process (e.g. `qemu-system-x86_64`) or docker processes only. QEMU processes or `virsh list --all` gives at least a hint that the virtualization KVM is running. What's inside those VMs is best to analyze when you log into the VM. KVM/libvirt (including QEMU) is only one virtualization technology under Linux using an own kernel. There's also Xen, VirtualBox and VMWare.

In any case it's important to find out whether the separation of the systems reflect their security contexts.

References

[1] Dirty COW, vulnerability and exploit from 2016

D06 - Protect Secrets

Passwords, private keys, tokens and similar sensible items granting access to your valuable assets must be protected at all time. The best is to encrypt it properly and then to store it with proper access rights.

Threat Scenarios

There are a variety of pitfalls one needs to avoid to not expose any secrets.

- Fictional scenario 1: Dockerfiles(or /YAML files) are used in company A to build containers. A junior developer was trying to accomplish a task where he wanted to deploy SSH keys into a container. Not only that running SSHD inside a container is a bad idea. He used the Dockerfile (or /YAML file) to reference

and he wasn't aware

- Dockerfile, public repo
- docker inspect
- Real world example [1]: IBM had an issue in 2017, where they left Docker TLS keys for the Swarm host API in the container. All what was needed to exploit this, was a web browser and a free account. (On top of it it was possible to spawn an arbitrary container to mount "/" from a host, yielding possibly to compromise the host).

How Do I prevent?

Protect the secrets by restricting access to them as good as possible and if it possible also encrypt the secrets properly. Depending on the secret both might not be doable at the same time.

- Environment variables passed to a docker process are inherited to each process running in the container. When an attacker is able to get any kind of access to a running process he is able to retrieve those credentials.
- Storing credentials in a Dockerfile is also a bad idea as it'll likely to be committed to a repository. Also if the repository is not public or if the Dockerfile might not be committed the information is hard to protect the secrets against further threats.
- Similar difficult: YAML file ### base64
- Storing credentials in the Docker image is a bad idea too. The registry where the image is stored might be either public and leak those credentials, or if it's not public it has unnecessarily loose access rights. *

There are also creative hacks out there in the internet which most of the time aren't secure either. Examples like curling or wgetting a secret, passing it to a service and remove it is also not a solution [2].

Except the mentioned real world incident regarding Tesla in the Threat Scenarios you can find a lot of bad examples out there in the internet. So part of the prevention is especially here having policies how to deal with different types of secrets and educate your developers.

In general this is a chicken and egg problem. If you have a key protected with a passphrase to protect another key, one needs to provide a passphrase.

Pass a pointer?

secret mount

How can I find out?

References

- [1] The Register: Big Blue's big blunder: IBM accidentally hands over root access to its data science servers. Details in wyc's domain: IBM Data Science Experience: Whole-Cluster Privilege Escalation Disclosure.
- [2] Worth reading is a thread from the Moby project.

D07 - Resource Protection

Threat Scenarios

Depending on the host OS there's no restriction at all for a container in terms of CPU, memory -- or network and disk I/O. The threat is that either due to a software failure or due to a deliberate cause by an attacker one of those resources runs short which affects physical resources of the underlying host and all other containers.

Also if a container has been contained by default security measures imposed by e.g. docker it still shares physical resources with other containers and the host, i.e. mostly CPU and memory. So if other containers use those resources extensively, there won't be much left for your container.

The network is also a shared medium and most likely when data is being read or written, it is the same resource.

For the memory it is important to understand that there's a so called OOM [1] killer in the host's Linux kernel. The OOM killer kicks in when the kernel is short of memory. Then it starts - using some algorithms [2] - to "free" memory so that the host and kernel itself can still survive. The processes being killed are not necessarily the ones to blame for the extensive memory consumption (OOM score see [3]) and often the host's RAM is oversubscribed [3].

How Do I prevent?

The best is first for containers to impose reasonable upper limits in terms of memory and CPU. By reaching those limits the container won't be able to allocate more memory or consume more CPU.

For memory there are two main variables for setting a hard limit `-memory` and `--memory-swap`. Soft limits can exceed the value specified. They would be set with `--memory-reservation`. For a more complete description see the docker documentation [4]. To protect processes in the container you can also set `--oom-kill-disable`. The container daemons have a lower OOM score and won't normally be killed).

How can I find out?

- Configured: `docker inspect`
- Live: `docker stats`, including what's configure
- Details memory: ``/sys/fs/cgroup/memory/docker/$CONTAINERID/*

References

- [1] OOM stands for Out of Memory Management
- [2] https://www.kernel.org/doc/gorman/html/understand/understand016.html
- [3] https://lwn.net/Articles/317814/
- [4] https://docs.docker.com/config/containers/resource_constraints/

D08 - Container Image Integrity and Origin

Threat Scenarios

How Do I prevent?

How can I find out?

References

D09 - Follow Immutable Paradigm

Threat Scenarios

How Do I prevent?

How can I find out?

References

D10 - Logging

Threat Scenarios

How Do I prevent?

How can I find out?

Moitor from the outside for open consoles

References

E11 - What's Next?

For Managers

For DevOps'

For Testers

(here latest mention CIS Benchmark)

References