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 [https://www.owasp.org/index.php/OWASP_Proactive_Controls OWASP Proactive Controls] to the [https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project 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.

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	Often the main security advantage of Docker is neglected: The application within the container runs with 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 the bugs. For all domains mentioned you need to decide when you apply regular and emergency patches.
D03 - Network Separation and Firewalling	You properly need to design your network upfront. Management interfaces from the orchestration tool as well as network services 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 a negative security impact.
D06 - Protect Secrets	Authentication and authorization of a microservice against a peer or a third party requires secrets to be provided. Also for an attacker those secrets potentially provide access to your data. 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 fully trustworthy, starting from the origin up until the deployment. You need to make sure that all transfers as well as the images at rest are secure.
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, not into the container.

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.

It should be noted that it is very dangerous for the host and all containers on this host to run a privileged container (`--privileged`) as it removes almost every restriction. Root in a container can access e.g. block devices, the /proc and /sys file system on the host and with a little work it can also load modules on the host [2].

How Do I prevent?

It is important to run your microservice with the least privilege possible. The good thing is that containers are unprivileged, unless you have configured them explicitly differently (e.g. `docker run --privileged`). However 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.

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

Please note that by patch management strategy (patch management plan or policy, or security SLA are used synonymous) in the following paragraphs the scope is not primarily a technical one: It's necessary to agree and have an approval when certain patches will be applied. Often this strategy would be a task of an Information Security Officer. But not having an ISO is no excuse for not having a patch management strategy.

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 important threats are kernel exploits from within a container through abuse of Linux sys(tem)calls which lead to root access. Also the orchestration software has interfaces which either maybe not be locked down [1] and have shown numerous problems in the past. e.g. like etcd [], kubelet [] and dashboard [].

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 risk is equal likelihood times damage. Which means also if you minimized the likelihood through partly mitigation or network access, the damage and thus the risk is very high. Patches make sure that the software is always as secure as provided from the vendor.

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

In general not patching in a timely fashion is the most frequent problem in the IT industry. Most software defects from "off the shelf software" are well known before exploits are written and used. Sometimes not even a sophisticated exploit is needed.

Same applies for your container environment. It is not as straight foward though as there are four different "patch domains":

- Images: the container operating 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.

Suggestion when to patch what

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

If you aren't in an environment which has change or patch policies or processes recommended is the following (also if you are it's recommended to review them whether it needs to be re-adjusted for you container environment):

- Define a time span in which outstanding patches will be applied on a regular basis. This can be different for each patch domain but it doesn't have to. Differences may arise due to different threat scenarios: An exposed container or API from your orchestration software which you need to expose has a higher threat level.
- Execute patch cycles and monitor them for success and failures.
- 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 can be as effective as not to patch. Those technical details which come after applying patches need to be defined in the the patch plan too.

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.

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

- `uptime`
- When were last patches applied (`rpm --qa --last` , `yum check-update` , `zypper lu` or check `/var/log/dpkg.log*` , `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 and you need to be sure that both patch management strategies exist and are being properly executed. Depending where you start with your policy recommended is [n-1]. [n].

References

Commercial

- [1] Lacework: Containers at Risk
- [n-1] TBD: ~~Good source (1) for patch management, lightweighted (not ITIL, nor ISO 2700x)~~
- [n] TBD: ~~Another good source (2) for patch management~~

D03 - Network Separation and Firewalling

In the old world one had a secured DMZ managed by an infrastructure or network team which made sure that the frontend server's service was in a locked down fashion reachable from the internet and e.g. can talk securely to the middleware and backend -- and to nothing else. Management interfaces from a serial console or a baseband management controller were put to a dedicated LAN with strict access controls.

This should be bascially your starting point when planning a network for your microservices.

Threat Scenarios

The container world changed the networking. The network is not necessarily divided into zones with strict firewall/routing rules. Without precautions it maybe even flat and every microservice is basically able to talk to all microservices, including interfaces of the management backplane - your orchestration tool or e.g. the host's services.

The paradigm having one microservice per containers makes matters not easier, as some microservices need to talk to each other while others should definitely not from the security standpoint.

Threats:

- Internet exposed management frontends/APIs from orchestration tool [1)]
- LAN/DMZ exposed management frontends/APIs from orchestration tool [1)]
- LAN/DMZ unnecessarily exposed microservices in the LAN from same application (token,)
- LAN/DMZ unnecessarily exposed classical services (NFS/Samba, CI/CD appliance, DBs)
- No 100% network separation between tenants as they share the same network
- Access to host network from a microservice

Except the first scenario: The threats are that an attacker got access to the local network (LAN/DMZ), e.g. though your compromised frontend service (internet) and moves from there around in this network.

How Do I prevent?

In one line: Have a mulitlayer network defense like in the old world and allow only a white-list based communication.

Do proper network planning:

- Choose the right network driver for your environment
- Segment your DMZ appropriately
- Muliple tenants should not share the same network
- Define necessary communication
- Protect management frontends/APIs. Never ever expose them in the internet.
- Also don't expose them in the DMZ. This is your management backplane. It needs strict white-list based network protection
- Protect the host via white-list

How can I find out?

If the network is segmented and not flat it is quite a task. Network information is best to request beforehand.

Just reading the output from host-based firewalls like `iptables -t nat -L -nv` and `iptables -L -nv` becomes a tedious task. The tool for network scanning and discovery is nmap [1]. If the network is not flat you probably want to scan from different source IPs. That could be hosts and/or a specially crafted docker container containing nmap. Depending on the capabilities preconfigured and whether TCP connect scans suffice you maybe need to allow nmap in a container to send raw packets (`cap_net_raw`).

References

[1] https://nmap.org/

---

1) That can be ones which require credentials or even not. Examples for both: etcd, dashboards, kubelet, ...

D04 - Secure Defaults and Hardening

While D03 - Network Separation and Firewalling for sure provides at least can an extra layer of protection for any network based services on the e.g. host, the orchestration tool and other places: it doesn't address the root cause. It mitigates the symptom. Best practice though is not to start any service which is not needed. And those services which are needed need to be locked 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, systemd-services
- Interfaces within the container, either from the microservice (e.g. spring-boot) or from the distribution.

How Do I prevent?

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. E.g. -- a standard Ubuntu system is a standard Ubuntu system. There might be other distributions more specialised on hosting containers. Then install a minimal distribution. Desktop applications, compiler environments or any server applications have no business here. They all add an attack surface. Think of a minimal bare metal system. When you pick an OS for the host, find out the support time left (EOL).

Also for the container, best practise is: do no install unnecessary packages [1]. Alpine Linux has a smaller footprint and has per default less binaries on board.

In general you need to make sure you know what services are offered from each component in your LAN. Then you need to decided 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 proper authentication needed 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 documention in terms of security?

For services which cannot be turned off, reconfigured or hardended: 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, do not switch those technologies off. Find the root causes in the system log file and relax those rule only.

How can I find out?

- System: Log into it with administrative privileges and see what's running using `netstat -tulp` or `lsof -i -Pn| grep -v ESTABLISHED`. This won't return the network sockets from the containers though.
- Container: Please note that `docker inspect` returns deliberately exposed ports only. **FIXME**

```
nmap -sTU -p1-65535 $(docker inspect $(docker ps -q) --format '\{\{ .NetworkSettings .IPAddress }}')
```

- As in D03 scanning the network would be another one option, albeit probably not as effective.

References

[1] Docker Best Practices

D05 - Maintain Security Contexts

Threat Scenarios

How Do I prevent?

How can I find out?

References

D06 - Protect Secrets

Threat Scenarios

How Do I prevent?

How can I find out?

References

D07 - Resource Protection

Threat Scenarios

How Do I prevent?

How can I find out?

References

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?

References