0.翻译人员

付强	周强强	杨川
冯其	刘锦	余章涛

一. 概述

本文档面向计划开发，部署，评估或保护包含Docker 1.13.0或更高版本技术解决方案的系统和 应用程序管理员，安全专家，审计人员，服务和平台部署人员。

1 目标读者

本文档——“CIS Docker 1.13.0基线” ——提供了有关为Docker容器版本1.13.0建立安全配置状态的说明性指导。 本指南在RHEL 7和Debian 8上针对Docker 1.13.0进行了测试。

2 共识指导

这个基线是由领域专家组成的共识评审流程创建的。 共识参与者提供来自不同背景的观点，包括咨询， 软件开发，合规审计，安全研究，运营，政府和法律。

每个CIS基线经历两个阶段的共识评审。第一阶段发生在初始基线开发阶段。 在这个阶段，领域专家召集讨论，创建和测试基线的草案。 这个讨论发生在基线建议达成共识之前。 第二阶段在基线发布后。 在这个阶段，互联网社区提供的所有反馈都由共识团队审查，以纳入基线。

3 评分说明

评分状态表示符合给定建议是否影响评估目标的基线评分。 这个基线使用以下得分状态：

评分状态	含义
Scored	不遵守 “**Scored**” 建议将会降低最终的基线分数。 遵守 “**Scored**” 建议将会增加最终的基线分数。
Not Socred	不遵守 “**Not Socred**” 建议不会降低最终的基线分数。 遵守 “**Not Socred**” 的建议不会增加最终的基线分数。

4 配置文件定义

基线定义以下配置文件：

- **Level 1- Docker**

  此配置文件中的项目：

  - 审慎务实；
  - 有明确的安全益处；
  - 不能以超出可接受的方式来限制技术的效用。

- **Level 1- Linux Host OS**

  此配置文件中的项目适用于Linux主机操作系统，同时：

  - 审慎务实；
  - 有明确的安全益处；
  - 不能以超出可接受的方式来限制技术的效用。

- **Level 2 - Docker**

  此配置文件中的项目具有以下一个或多个特征：

  - 旨在用于安全性至关重要的环境或用例；
  - 作为纵深防御措施；
  - 可能会负面地抑制该技术的使用或性能。

二. 推荐基线

1 主机配置

本节将介绍为用于执行容器化工作负载主机应遵循的安全建议。

保护Docker主机并遵循基础设施安全最佳实践将为执行容器化工作负载的主机构建坚实可靠的基础。

1.1 为容器创建一个单独的分区（scored）

配置适用性：

- **Level 1- Docker**

描述

所有Docker容器及其数据和元数据都存储在`/ var / lib / docker`目录下。 默认情况下，`/ var / lib / docker`将根据可用性挂载在`/`或`/ var`分区下。

缘由

Docker 将`/ var / lib / docker`作为存储所有Docker相关文件（包括 images）的默认目录。 这个目录可能会快速填满，Docker和主机将变的无法使用。 所以，建议创建一个单独的分区（逻辑卷）来存储Docker文件。

审计

在Docker主机上执行以下命令：

```
grep /var/lib/docker /etc/fstab
```

这应该返回`/ var / lib / docker`挂载点分区的详细信息。

修正

对于新安装的系统，为`/ var / lib / docker`挂载点创建一个单独的分区。 对于已安装的系统，请使用逻辑卷管理器（LVM）创建分区。

影响

无

默认值

默认情况下，`/ var / lib / docker`将根据可用性安装在`/`或`/ var`分区下。

参考文献

```
1. http://www.projectatomic.io/docs/docker-storage-recommendation
```

1.2 加强容器主机（Not Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

容器在Linux主机上运行。 容器主机可以运行一个或多个容器。 加强主机安全以减轻主机安全配置错误是非常重要的。

缘由

应该遵循基础设施安全最佳实践并加强您的主机操作系统。 强化主机系统将使得主机漏洞得到缓解。 否则，可能会导致安全风险和违规行为。

审计

确保遵循主机特定的安全准则。 向系统管理员询问当前主机系统符合哪个安全基线。 确保主机系统实际上符合主机特定的安全基线。

修正

可以考虑您的容器主机的各种CIS安全基线。 如果您有其他安全指导方针或法规要求要遵守，请在您的环境中按照适当的方式进行操作。 另外，你可以使用 `grsecurity` 和 `PaX` 运行一个内核。 这会在编译时和运行时增加许多安全检查。 它也被设计用于防御许多漏洞，具有强大的安全功能。 这些功能不需要特定于Docker的配置，因为这些安全功能适用于系统范围，与容器无关。

影响

无

默认值

默认情况下，主机具有出厂设置。 它没有强化。

参考文献

```
1. https://docs.docker.com/articles/security/
2. https://benchmarks.cisecurity.org/downloads/multiform/index.cfm
3. http://docs.docker.com/articles/security/#other-kernel-security-features
4. https://grsecurity.net/
5. https://en.wikibooks.org/wiki/Grsecurity
6. https://pax.grsecurity.net/
7. http://en.wikipedia.org/wiki/PaX
```

1.3 保持Docker 及时更新（Not Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

Docker软件频繁发布，可解决安全漏洞，产品缺陷和引入新功能。 在这些产品更新中保留了一个选项卡，并在新的安全漏洞被修复或被认为是正确的时候会频繁升级。

缘由

通过及时进行Docker更新，Docker软件中的漏洞可以得到缓解。 受过教育的攻击者可能会在尝试访问或提升权限时利用已知的漏洞。 不安装常规的Docker更新可能会让您运行易受攻击的Docker软件。 这可能会导致权限提升，未经授权的访问或其他安全漏洞。 跟踪新版本并根据需要进行更新。

审计

执行以下命令并验证Docker版本是否为最新版本。 尽管这不是最新的版本。

```
docker version
```

修正

跟踪Docker版本并根据需要进行更新。

影响

无

默认值

不适用

参考文献

```
1. https://docs.docker.com/installation/
2. https://github.com/docker/docker/releases/latest
```

1.4 只允许可信用户控制Docker守护进程（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

Docker守护进程当前需要`“root”`权限。 添加到 `“docker”` 组的用户拥有充分的 `“root”`访问权限。

缘由

Docker允许在Docker主机和访客容器之间共享一个目录，而不会限制容器的访问权限。 这意味着您可以启动一个容器并将主机上的 `“/”` 目录映射到容器。 容器将能够不受任何限制地更改您的主机文件系统。 简而言之，这意味着只要是 `“docker”` 组的成员，然后在主机上启动一个映射 `/` 目录的容器，就可以获得更高的权限。

审计

在docker主机上执行以下命令，并确保只有受信任的用户是`“docker”` 组的成员。

```
getent group docker
```

修正

从 `“docker”` 组中删除任何不受信任的用户。 此外，不要创建主机上的敏感目录到容器卷的映射。

影响

以普通用户身份建立和执行容器的权限将受到限制。

默认值

不适用

参考文献

```
1. https://docs.docker.com/articles/security/#docker-daemon-attack-surface
2. https://www.andreas-jung.com/contents/on-docker-security-docker-group-considered-harmful
3. http://www.projectatomic.io/blog/2015/08/why-we-dont-let-non-root-users-run-docker-in-centos-fedora-or-rhel/
```

1.5 审计docker daemon（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

审计所有的Docker daemon活动。

缘由

除了审计常规的Linux文件系统和系统调用外，还要审计Docker daemon。 Docker daemon以 `“root”` 权限运行。 因此有必要审计其活动和使用情况。

审计

验证是否有Docker daemon 的审计规则。 例如，执行以下命令：

```
auditctl -l | grep /usr/bin/docker
```

这应该列出Docker守护进程的规则。

修正

为Docker daemon 添加一条规则。 例如，在 `/etc/audit/audit.rules` 文件中添加如下行：

```
-w /usr/bin/docker -k docker
```

然后，重新启动审计 daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，Docker daemon不被审计。

参考文献

```
1. https://access.redhat.com/documentation/en-
US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
```

1.6 审计Docker文件和目录 - / var / lib / docker（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

审计 `/var/lib/docker`

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon以 `“root”` 权限运行。 它的行为取决于一些关键的文件和目录。 `/ var / lib / docker` 就是一个这样的目录， 它包含有关容器的所有信息。 它必须被审计。

审计

验证是否存在与 `/ var / lib / docker` 目录相对应的审计规则。

例如，执行以下命令：

```
auditctl -l | grep /var/lib/docker
```

这应该列出 `/ var / lib / docker` 目录的规则。

修正

为 `/ var / lib / docker` 目录添加一条规则。

例如， 在/etc/audit/audit.rules文件中添加如下行：

```
-w /var/lib/docker -k docker
```

然后，重新启动审计 daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。

参考文献

```
 1.  https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
```

1.7 审计Docker文件和目录 - / etc /docker（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

审计 `/etc/docker`。

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker deaemon 以 `“root”`权限运行。 它的行为取决于一些关键的文件和目录。 `/ etc / docker` 就是一个这样的目录。 它拥有用于Docker daemon和Docker客户端之间TLS通信的各种证书和密钥。 它必须被审计。

审计

验证是否存在与 `/ etc / docker` 目录相对应的审计规则。

例如，执行以下命令：

```
auditctl -l | grep /etc/docker
```

这应该列出 `/ etc / docker` 目录的规则。

修正

为 `/ etc / docker` 目录添加一条规则。

例如， 在 `/etc/audit/audit.rules` 文件中添加如下行：

```
-w /etc/docker -k docker
```

然后，重新启动审计 daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。

参考文献

```
 1.  https://access.redhat.com/documentation/en-
US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
```

1.8 审计Docker文件和目录 - docker.service（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

如果可以，审计 `docker.service`

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon 以 `“root”` 权限运行。 它的行为取决于一些关键的文件和目录。 `docker.service` 就是这样一个文件。 如果daemon 参数已被管理员更改，则 `docker.service` 文件可能存在。 它拥有Docker daemon的各种参数。 如果可用，它必须被审计。

审计

步骤1：找出文件位置：

```
systemctl show -p FragmentPath docker.service
```

步骤2：如果该文件不存在，则该基线不适用。 如果文件存在，请验证是否存在与该文件相对应的审计规则：

例如，执行以下命令：

```
auditctl -l | grep docker.service
```

这应该根据其位置列出 `docker.service` 的规则。

修正

如果该文件存在，请为其添加规则。

例如， 在 `/etc/audit/audit.rules` 文件中添加如下行：(注意：下面的路径根据实际改变)

```
-w /usr/lib/systemd/system/docker.service -k docker
```

然后，重新启动审计 daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。 `docker.service` 文件可能在系统上不可用。

参考文献

```
 1.  https://access.redhat.com/documentation/en-
US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
```

1.9 审计Docker文件和目录 - docker.socket（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

如果可以，审计 `docker.socket` 。

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon以 `“root”` 权限运行。 它的行为取决于一些关键的文件和目录。 `docker.socket` 就是这样一个文件。 它拥有Docker daemon socket的各种参数。 如果可以，它必须被审计。

审计

步骤1：找出文件位置：

```
systemctl show -p FragmentPath docker.socket
```

步骤2：如果该文件不存在，则该基线不适用。 如果文件存在，请验证是否存在与该文件相对应的审计规则：

例如，执行以下命令：

```
auditctl -l | grep docker.socket
```

这应该根据其位置列出 `docker.socket` 的规则。

修正

如果该文件存在，请为其添加规则。

例如， 在 `/etc/audit/audit.rules` 文件中添加如下行：(注意：下面的路径根据实际改变)

```
-w /usr/lib/systemd/system/docker.socket -k docker
```

然后，重新启动审计 daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。 `docker.socket` 文件可能在系统上不可用。

参考文献

```
 1.  https://access.redhat.com/documentation/en-
US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
```

1.10 审计Docker文件和目录 - /etc/default/docker（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

如果可以，审计 `/etc/default/docker`。

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon 以`“root”`权限运行。 它的行为取决于一些关键的文件和目录。`/ etc / default / docker`就是这样一个文件。 它拥有Docker daemon的各种参数。 如果可以，它必须被审计。

审计

验证是否存在与`/ etc / default / docker`文件相对应的审计规则。

例如，执行以下命令：

```
auditctl -l | grep /etc/default/docker
```

这应该列出`/ etc / default / docker`文件的规则。

修正

为`/ etc / default / docker`文件添加一条规则。

例如， 在`/etc/audit/audit.rules`文件中添加如下行：

```
-w /etc/default/docker -k docker
```

然后，重新启动审计daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。 文件`/ etc / default / docker`可能在系统上不可用。 在这种情况下，这个推荐基线是不适用的。

参考文献

```
1.  https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
```

1.11 审计Docker文件和目录 - /etc/docker/daemon.json（Scored）

配置适用性

- Level 1- Linux Host OS

描述

如果可以，审计 `/etc/docker/daemon.json`。

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon以 `“root”`权限运行。 它的行为取决于一些关键的文件和目录。 `/etc/docker/daemon.json` 就是这样一个文件。 它拥有Docker daemon的各种参数。 如果可以，它必须被审计。

审计

验证是否存在与 `/etc/docker/daemon.json`文件相对应的审计规则。

例如，执行以下命令：

```
auditctl -l | grep /etc/docker/daemon.json
```

这应该列出 `/etc/docker/daemon.json` 文件的规则。

修正

为 `/etc/docker/daemon.json` 文件添加一条规则。

例如， 在 `/etc/audit/audit.rules`文件中添加如下行：

```
-w /etc/docker/daemon.json -k docker
```

然后，重新启动审计daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。 文件 `/etc/docker/daemon.json` 可能在系统上不可用。 在这种情况下，这个推荐基线是不适用的。

参考文献

```
1. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
2. https://docs.docker.com/engine/reference/commandline/daemon/#daemon-configuration-file
```

1.12 审计Docker文件和目录 - /usr/bin/docker-containerd（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

如果可以，审计 `/usr/bin/docker-containerd`。

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon以 `“root”` 权限运行。 它的行为取决于一些关键的文件和目录。 `/usr/bin/docker-containerd` 就是这样一个文件。 Docker现在依靠 `containerd` 和 `runC` 来产生容器。 如果可以，它必须被审计。

审计

验证是否存在与 `/usr/bin/docker-containerd` 文件相对应的审计规则。

例如，执行以下命令：

```
auditctl -l | grep /usr/bin/docker-containerd
```

这应该列出 `/usr/bin/docker-containerd` 文件的规则。

修正

为 `/usr/bin/docker-containerd` 文件添加一条规则。

例如， 在 `/etc/audit/audit.rules` 文件中添加如下行：

```
-w /usr/bin/docker-containerd -k docker
```

然后，重新启动审计daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。 文件 `/usr/bin/docker- containerd` 可能在系统上不可用。 在这种情况下，这个推荐基线是不适用的。

参考文献

```
1. https://access.redhat.com/documentation/en-
US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
2. https://github.com/docker/docker/pull/20662
3. https://containerd.tools/
```

1.13 审计Docker文件和目录 - /usr/bin/docker-runc（Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

如果可以，审计 `/usr/bin/docker-runc` 。

缘由

除了审计您的常规Linux文件系统和系统调用外，还要审计所有Docker相关的文件和目录。 Docker daemon以 `“root”` 权限运行。 它的行为取决于一些关键的文件和目录。 `/usr/bin/docker-runc` 就是这样一个文件。 Docker现在依靠 `containerd` 和 `runC` 来产生容器。 如果可以，它必须被审计。

审计

验证是否存在与 `/usr/bin/docker-runc` 文件相对应的审计规则。

例如，执行以下命令：

```
auditctl -l | grep /usr/bin/docker-runc
```

这应该列出 `/usr/bin/docker-runc` 文件的规则。

修正

为 `/usr/bin/docker-runc` 文件添加一条规则。

例如， 在`/etc/audit/audit.rules`文件中添加如下行：

```
-w /usr/bin/docker-runc -k docker
```

然后，重新启动审计daemon。 例如，

```
service auditd restart
```

影响

审计会生成相当大的日志文件。 确保定期转移和归档。 此外，创建一个单独的审计分区，以避免填充根文件系统。

默认值

默认情况下，不会审计Docker相关的文件和目录。 文件 `/usr/bin/docker- runc` 可能在系统上不可用。 在这种情况下，这个建议是不适用的。

参考文献

```
1. https://access.redhat.com/documentation/en-
US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html
2. https://github.com/docker/docker/pull/20662
3. https://containerd.tools/
4. https://github.com/opencontainers/runc
```

2 Docker daemon 配置

本节列出了修改、增强Docker daemon(服务器端)行为的推荐基线。以下的设置会影响所有容器。

注意：Docker daemon 的可选配置可以使用Debian和Ubuntu下的/etc/sysconfig/docker或/etc/default/docker文件来进行配置。同时，docker daemon 模式与 `/user/bin/dockerd`， `docker` 服务的 `-d` 或 `daemon` 参数等价。

2.1 限制容器间(Scored)

配置适用性

- **Level 1- Docker**

描述

默认情况下，在同一主机上的容器是允许互相之间通信的。如果不想，限制所有容器间的通信。在需要容器间通讯时，将相关的容器 link 起来。

缘由

默认情况下，在同一主机上的所有容器之间使用不受限制的网络通信。 因此，每个容器都有可能读取同一主机容器网络上的所有数据包。 这可能会导致意外和不必要的将信息泄露给其他容器。 因此，限制容器间的通信。

审计

```
ps -ef | grep dockerd
```

确保 `'--icc'` 参数被设置为 `'false'`，或者通过如下命令确认。

```
docker network ls --quiet | xargs xargs docker network inspect --format '{{
.Name }}:{{ .Options }} '
```

将会返回：`com.docker.network.bridge.enable_icc:false`。

修正

以`'--icc=false'` 参数运行docker daemon。

例如：

```
/usr/bin/dockerd --icc=false
```

影响

容器间通信将被禁用。 容器将不能与同一主机上的另一个容器进行通信。 如果需要在同一主机上的容器之间进行通信，则需要使用容器链接来显式定义它。

默认值

默认情况下，是允许容器间通信的。

参考文献

```
1. https://docs.docker.com/articles/networking
```

2.2 设置logging级别(Scored)

配置适用性

- **Level 1- Docker**

描述

设置Docker daemon的日志级别为 `'info'`

缘由

设置适当的日志级别，配置Docker daemon 以记录您稍后需要查看的事件。将捕获除调试日志以外，`“info”`及其以上的基本日志级别的所有日志。 除非需要，否则不应在`“debug”`日志级别运行Docker daemon。

审计

```
ps -ef | grep dockerd
```

确保没有`'--log-level'`参数，如果有，修改为`'info'`级别。

修正

以如下方式运行 docker daemon：

```
dockerd --log-level="info"
```

影响

无

默认值

默认情况下，Docker daemon 的日志级别为 `'info'`。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/
```

2.3 允许Docker改变iptables(Scored)

配置适用性

- **Level 1- Docker**

描述

`Iptables`是用于设置，维护和检查Linux内核中的IP包过滤规则表。 允许Docker daemon 更改`iptables`。

缘由

如果您选择这么做，Docker也不会更改您系统的 `iptables`。 Docker 服务会自动根据您如何选择容器的网络选项（如果允许的话）对`iptables`进行必要的修改。 建议让Docker 服务自动更改`iptables`，以避免可能妨碍容器与外界通信的网络配置错误。 此外，每次选择运行容器或修改网络选项时，都可以减少更新 `iptables` 的麻烦。

审计

```
ps -ef | grep dockerd
```

确保`“–iptables”`参数不存在或未设置为`“false”`。

修正

不要使用`“--iptables = false”`参数运行Docker daemon。

例如，不要以如下方式启动Docker daemon：

```
dockerd --iptables=false
```

影响

无

默认值

默认情况下，`“iptables”`设置为`“true”`。

参考文献

```
1. https://docs.docker.com/v1.8/articles/networking/
```

2.4 不使用不安全的镜像仓库(Scored)

配置适用性

- **Level 1- Docker**

描述

Docker认为私有仓库是安全的或不安全的。 默认情况下，仓库被认为是安全的。

缘由

一个安全的仓库使用TLS。 仓库的CA证书副本放置在Docker主机的`'/etc/docker/certs.d/<registry-name>/'`目录中。 不安全的仓库是没有有效的仓库证书或没有使用TLS。 您不应该在生产环境中使用任何不安全的仓库。 不安全的注册仓库可能会被篡改，从而导致生产系统受到影响。 此外，如果一个仓库被标记为不安全，那么`'docker pull'`,`'docker push'`和`'docker search'`命令将不会产生错误消息，用户可能会无限期地使用不安全的仓库，而不会得知潜在的危险。

审计

运行`docker info`或者执行下面的命令来查看是否使用了不安全的仓库：

```
ps -ef | grep dockerd
```

确保`'--insecure-registry'`参数不存在。

修正

不要使用任何不安全的仓库。 例如，不要以如下方式启动Docker daemon：

```
dockerd --insecure-registry 10.1.0.0/16
```

影响

无

默认值

默认情况下，Docker假定所有本地的仓库都是安全的。

参考文献

```
1. https://docs.docker.com/registry/insecure/
```

2.5 不要使用aufs存储驱动(Scored)

配置适用性

- **Level 1- Docker**

描述

不要使用`'aufs'`作为你的Docker实例的存储驱动。

缘由

`'aufs'`存储驱动程序是最旧的存储驱动程序。 它基于Linux内核补丁集，不太可能被合并到主要的Linux内核中。 `“aufs”`驱动程序会导致一些严重的内核崩溃。 `'aufs'`只有Docker的传统支持。 最重要的是，在许多使用了最新Linux内核的Linux发行版中，`“aufs”`不再是受支持的驱动程序。

审计

执行以下命令并确认`“aufs”`不被用作存储驱动程序：

```
docker info | grep -e "^Storage Driver:\s*aufs\s*$"
```

上面的命令不应该返回任何东西。

修正

不要明确地使用`“aufs”`作为存储驱动程序。 例如，不要以如下方式启动Docker守护进程：

```
dockerd --storage-driver aufs
```

影响

`'aufs'`是唯一允许容器共享可执行文件和共享库内存的存储驱动程序。 如果您使用相同的程序或库运行数千个容器，这可能会很有用。

默认值

默认情况下，Docker在大多数平台上使用 `“devicemapper”`作为存储驱动程序。 默认存储驱动程序可以根据您的操作系统供应商而有所不同，您应该使用您的供应商支持的首选存储驱动程序。

参考文献

```
1. http://docs.docker.com/reference/commandline/cli/#daemon-storage-driver-
option
2. http://muehe.org/posts/switching-docker-from-aufs-to-devicemapper/
3. http://jpetazzo.github.io/assets/2015-03-05-deep-dive-into-docker-
storage-
drivers.html#1
4. https://docs.docker.com/engine/userguide/storagedriver/
```

2.6 为Docker daemon 配置TLS认证(Scored)

配置适用性

- **Level 1- Docker**

描述

可以让Docker daemon 监听特定的IP和端口以及任何其他Unix 非默认套接字。 配置TLS身份验证，以限制访问Docker daemon 的IP和端口。

缘由

默认情况下，Docker daemon绑定到非网络连接的Unix套接字并以 `“root”`权限运行。 如果将默认docker daemon绑定到TCP端口或任何其他Unix套接字，那么任何有权访问该端口或套接字的人都可以完全访问Docker守护进程，并转而访问主机系统。 因此，不应该将Docker daemon绑定到另一个IP /端口或Unix套接字。 如果您必须通过网络套接字公开Docker daemon，请为daemon 和Docker Swarm API（如果使用）配置TLS身份验证。 这将限制通过网络到Docker daemon的连接，将通过TLS成功进行身份验证的客户端限制在一定数量。

审计

```
ps -ef | grep dockerd
```

确保以下参数存在：

- `'–tlsverify'`
- `'–tlscacert'`
- `'–tlscert'`
- `'–tlskey'`

修正

遵循Docker文档或其他参考中提到的步骤。

影响

您需要管理和保护Docker daemon和Docker客户端的证书和密钥。

默认值

默认情况下，没有配置TLS认证。

参考文献

```
1. http://docs.docker.com/articles/https/
```

2.7根据需要设置默认的ulimit（Not Scored）

配置适用性

- **Level 1- Docker**

描述

在您的环境中根据需要设置默认的ulimit选项。

缘由

`ulimit` 提供了对shell可用的资源以及由它启动的进程的控制。 合理地设置系统资源限制可以使您免受许多灾难的困扰，例如，fork 炸弹。 有时候，即使友好的用户和合法的进程也会过度使用系统资源，从而导致系统无法使用。 设置Docker daemon的默认`ulimit`将强制所有容器实例的ulimit。 你不需要为每个容器实例设置 `ulimit`。 但是，如果需要，默认的 `ulimit`可以在容器运行时重写。 因此，要控制系统资源，请在您的环境中根据需要定义默认的`ulimit`。

审计

```
ps -ef | grep dockerd
```

确保`“--default-ulimit”`参数被设置为合适的值。

修正

在daemon 模式下运行docker，并在您的环境中根据相应的ulimits传递`“--default-ulimit”`作为参数。

例如，

```
dockerd --default-ulimit nproc=1024:2408 --default-ulimit nofile=100:200
```

影响

如果 `ulimits` 设置不正确，可能无法实现所需的资源控制，甚至可能导致系统无法使用。

默认值

默认情况下，没有设置ulimit。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/#default-ulimits
```

2.8启用用户名称空间支持（Scored）

配置适用性

- **Level 2- Docker**

描述

在Docker daemon中启用用户名称空间以利用容器用户与主机用户的重新映射。 如果您正在使用的容器没有在容器镜像中定义明确的容器用户，则此建议是有益的。 如果您正在使用的容器镜像具有预定义的非root用户，则可以跳过此建议，因为此功能仍处于初始阶段，可能会给您带来不可预知的问题和复杂性。

缘由

Docker daemon中的Linux内核用户名称空间为Docker主机系统提供了额外的安全性。 它允许容器具有，在主机系统使用的传统用户和组范围之外的，唯一范围的用户和组ID。 例如， `root` 用户将在容器内具有预期的管理权限，但可以有效地映射到主机系统上的非特权UID。

审计

```
ps -p $(docker inspect --format='{{ .State.Pid }}' <CONTAINER ID>) -o pid,user
```

以上命令将查找容器的PID，然后列出与容器进程关联的主机用户。 如果容器进程以root用户身份运行，不符合此基线。 或者，您可以运行 `docker info` 以确保在 `Security Options` 下列出 `userns` ：

```
docker info --format '{{ .SecurityOptions }}'
```

修正

请参阅Docker文档，了解可以根据您的要求配置的各种方法。 您的步骤也可能因平台而异 。例如，在Red Hat上，sub-UID和 sub-GID映射创建不能自动工作。 您可能需要创建自己的映射。

但是，高层次步骤如下：

步骤1：确保文件 `/ etc / subuid` 和 `/ etc / subgid` 存在。

```
touch /etc/subuid /etc/subgid
```

步骤2：用`--userns-remap`标志启动docker daemon。

```
dockerd --userns-remap=default
```

影响

用户命名空间重新映射使得不少Docker功能不兼容，而且目前也打破了一些功能。 查看Docker文档和参考链接了解详细信息。

默认值

默认情况下，用户命名空间不会被重新映射。

参考文献

```
1. http://man7.org/linux/man-pages/man7/user_namespaces.7.html
2. https://docs.docker.com/engine/reference/commandline/daemon/
3.
http://events.linuxfoundation.org/sites/events/files/slides/User%20Namespace
s
%20-%20ContainerCon%202015%20-%2016-9-final_0.pdf
4. https://github.com/docker/docker/issues/21050
```

2.9 确认默认cgroup使用情况（Scored）

配置适用性

- **Level 2- Docker**

描述

`--cgroup-parent`选项允许您为所有容器设置默认的cgroup父项。 如果没有特定用例，则应将此设置保留为默认值。

缘由

系统管理员通常定义容器应该在哪个`cgroup`下运行。 即使 cgroups没有被系统管理员明确定义，默认情况下，容器会在`docker` cgroup下运行。 这是可能的附加到一个不同于默认的cgroup。 这个用法应该被监视和确认。 通过附加到不同于默认的cgroup，可以不均匀地共享资源，从而可能导致主机资源匮乏。

审计

```
ps -ef | grep dockerd
```

确保`“--cgroup-parent”`参数未设置，或者设置为非默认cgroup。

修正

默认设置足够好，可以保持原样。 如果你想专门设置一个非缺省的cgroup，在启动时将`--cgroup-parent`参数传递给docker daemon。 例如，

```
dockerd --cgroup-parent=/foobar
```

影响

无

默认值

默认情况下，docker daemon为 fs cgroup驱动程序使用 `/docker`，为 systemd cgroup驱动程序使用 `system.slice`。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/
```

2.10 在需要时才改变基础设备大小（Scored）

配置适用性

- **Level 2- Docker**

描述

在某些情况下，您可能需要容量大于10G的容器。 在这些情况下，请仔细选择基础设备大小。

缘由

在 daemon 重启时，可以增加基本设备的大小。 增加基础设备尺寸允许所有将来的镜像和容器成为新的基础设备尺寸。 用户可以使用此选项来扩展基础设备尺寸，但不允许收缩。 该值影响可能已经被初始化并被拉取的镜像继承的系统范围的“基本”空文件系统。 虽然如果文件系统为空，文件系统不会分配增加的大小，但根据设备的大小，它将占用更多的空间。 这可能会导致文件系统被过度分配或已满而导致拒绝服务。

审计

```
ps -ef | grep dockerd
```

执行上面的命令，它不应该显示任何`--storage-opt dm.basesize`参数。

修正

直到需要时，才设置 `--storage-opt dm.basesize`。

影响

无

默认值

默认的基础设备大小是10G。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/#storage-driver-options
```

2.11 使用授权插件（Scored）

配置适用性

- **Level 2- Docker**

描述

使用授权插件来管理对Docker daemon的访问。

缘由

Docker的开箱即用授权模式是全有或者全无。 任何有权访问Docker daemon的用户都可以运行任何Docker客户端命令。 对于使用Docker的远程API来调用 daemon 的调用者也是如此。 如果您需要更大的访问控制，您可以创建授权插件并将其添加到您的Docker daemon 配置。 使用授权插件，Docker管理员可以配置访问策略粒度来管理对Docker daemon的访问。

审计

```
ps -ef | grep dockerd
```

确保 `“--authorization-plugin”` 参数被设置为合适的值。

修正

步骤1：安装/创建授权插件； 步骤2：根据需要配置授权策略。 步骤3：如下启动docker daemon：

```
dockerd --authorization-plugin=<PLUGIN_ID>
```

影响

每个docker命令专门通过授权插件机制。 这可能会导致性能下降。

默认值

默认情况下，授权插件没有设置。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/#access-authorization
2. https://docs.docker.com/engine/extend/authorization/
3. https://github.com/twistlock/authz
```

2.12 配置集中和远程日志（Scored）

配置适用性

- **Level 2- Docker**

描述

Docker现在支持各种日志驱动程序。 存储日志的最佳方式是支持集中式和远程日志。

缘由

集中和远程日志确保所有重要的日志都是安全的，尽管发生灾难性事件。 Docker现在支持各种各样的日志驱动程序。 使用最适合你的环境的那个。

审计

运行 `docker info` 并确保根据需要设置 `Logging Driver` 属性。

```
docker info --format '{{ .LoggingDriver }}'
```

或者，如果配置的话， 下面的命令会给你 `--log-driver` 设置，确保它是适当的值。

```
ps -ef | grep dockerd
```

修正

步骤1：按照其文档设置所需的日志驱动程序。

步骤2：使用该日志驱动程序启动docker守护进程。

例如，

```
dockerd --log-driver=syslog --log-opt syslog-address=tcp://192.xxx.xxx.xxx
```

影响

无

默认值

默认情况下，容器日志被保存为json文件

参考文献

```
1. https://docs.docker.com/engine/admin/logging/overview/
```

2.13 禁用旧版仓库的操作（v1）（Scored）

配置适用性

- **Level 1- Docker**

描述

最新的Docker 仓库是v2。 旧仓库版本（v1）上的所有操作都应受到限制。

缘由

Docker 仓库v2相比于v1，带来了许多性能和安全性方面的改进。 它支持容器镜像来源以及其他安全功能，如镜像签名和验证。 因此，Docker遗留仓库的操作应该受到限制。

审计

```
ps -ef | grep dockerd
```

上面的命令应该列出`--disable-legacy-registry`传递给docker守护进程的选项。

修正

以如下方式启动docker daemon：

```
dockerd --disable-legacy-registry
```

影响

旧版仓库的操作将受到限制。

默认值

默认情况下，旧版仓库操作是允许的。

参考文献

1. https://docs.docker.com/engine/reference/commandline/daemon/
2. https://github.com/docker/docker/issues/8093
3. https://github.com/docker/docker/issues/9015
4. https://github.com/docker/docker-registry/issues/612
5. https://docs.docker.com/registry/spec/api/
6. https://the.binbashtheory.com/creating-private-docker-registry-2-0-with-token-
authentication-service/
7. https://blog.docker.com/2015/07/new-tool-v1-registry-docker-trusted-registry-
v2-open-source/
8. http://www.slideshare.net/Docker/docker-registry-v2

2.14 启用实时还原（Scored）

配置适用性

- **Level 1- Docker**

描述

`'--live-restore'`可以在docker中完全支持 daemon-less 的容器。 它确保docker不会在关闭或恢复时停止容器，并在重新启动时正确重新连接到容器。

缘由

其中一个重要的安全特性是可用性。 在docker daemon中设置`'--live-restore'`标志可以确保当docker daemon 不可用时容器的执行不会中断。 这也意味着现在可以在不停机的情况下，更容易更新和修补docker daemon 。

审计

运行 `Docker info` 和确保`Live Restore Enabled`属性设置为 `true`。

```
docker info --format '{{ .LiveRestoreEnabled }}'
```

或者运行下面的命令，并确保使用 `--live-restore`。

```
ps -ef | grep dockerd
```

修正

以 `'--live-restore'` 运行docker daemon。

例如，

```
dockerd --live-restore
```

影响

无

默认值

默认情况下，没有启用`--live-restore`。

参考文献

```
1. https://github.com/docker/docker/pull/23213
```

2.15 在不需要swarm 时，不要启用它（Scored）

配置适用性

- **Level 1- Docker**

描述

除非需要，否则不要在Docker引擎实例上启用 swarm 模式。

缘由

默认情况下，Docker引擎实例不会侦听任何网络端口，所有与客户端的通信都通过Unix套接字进行。在Docker引擎实例上启用Docker swarm模式时，系统将打开多个网络端口，并将其提供给网络中的其他系统以用于群集管理和节点通信。 打开系统上的网络端口会增加攻击面，除非需要，否则应该避免这种情况。

审计

查看`docker info`命令的输出。 如果输出包括`Swarm: active`，则表示在Docker引擎上已经激活了 swarm 模式。 确认Docker引擎实例上是否需要swarm模式。

修正

如果 swarm 模式在系统上出错，请运行

`docker swarm leave`

影响

无

默认值

默认情况下，没有启用Docker swarm 模式。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/swarm_init/
```

2.16 控制swarm 中管理节点的数量（Scored）

配置适用性

- **Level 1- Docker**

描述

确保 swarm 创建所需管理节点的最小数量。

缘由

swarm 中的管理节点可以控制swarm，并修改其以修改安全参数的配置。 拥有过多的管理节点会使 swarm 更容易受到损害。 如果管理节点中不需要容错功能，则应当选择一个节点作为管理者。 如果需要容错，则应该配置最小的实际奇数以达到适当的容错水平。

审计

运行 `docker info` 并验证管理节点的数量。

```
docker info --format '{{ .Swarm.Managers }}'
```

或者运行下面的命令。

```
docker node ls | grep 'Leader'
```

修正

如果配置的管理节点数量过多，则可以使用以下命令将多余的工人降级：

```
docker node demote <ID>
```

其中 `<ID>` 是要降级节点的ID值。

影响

无

默认值

一个管理节点就是启动一个给定集群所需要的一切。

参考文献

```
1. https://docs.docker.com/engine/swarm/manage-nodes/
2. https://docs.docker.com/engine/swarm/admin_guide/#/add-manager-nodes-for-fault-tolerance
```

2.17 将 swarm 服务绑定到特定的主机接口（Scored）

配置适用性

- **Level 1- Docker**

描述

默认情况下，Docker swarm 服务将侦听主机上的所有接口，这对于主机具有多个网络接口的 swarm 的操作可能不是必需的。

缘由

当swarm 初始化时，`--listen-addr`标志的缺省值是`0.0.0.0:2377`，这意味着swarm服务将监听主机上的所有接口。 如果主机具有多个网络接口，则这可能是不合需要的，因为它可能将 docker swarm服务暴露给不涉及swarm 操作的网络。 通过将特定的IP地址传递给 `--listen-addr`，可以指定一个特定的网络接口来限制这种暴露。

审计

在端口2377 / TCP上列出网络监听器（docker swarm的默认设置），并确认它只监听特定的接口。 例如，使用ubuntu可以使用以下命令完成：

```
netstat -lt | grep -i 2377
```

修正

对此进行修复需要重新初始化swarm ，以指定`--listen-addr`参数的特定接口。

影响

无

默认值

默认情况下，docker swarm服务监听所有可用的主机接口。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/swarm_init/#/listen-addr-value
2. https://docs.docker.com/engine/swarm/admin_guide/#/recover-from-disaster
```

注意：我注意到看这个的几个观点。 docker swarm update似乎没有更改listen-addr参数。 对于补救措施，我确实想知道是否可以使用 -- force-new-swarm 改变这种情况，但是我不确定对swarm会有什么其他的影响，因此只剩下一般要求来重新初始化swarm。 另外有趣的是，在7946 / TCP上运行的节点通信服务并不遵守--listen-addr参数。 这对我来说似乎是一个错误，我可能会在github上进行更多的探索之后提出一个问题。

2.18 禁用用户级代理（Scored）

配置适用性

- **Level 1- Docker**

描述

docker daemon在端口暴露时启动用于端口转发的用户级代理服务。 在 hairpin NAT可用的地方，这种服务通常是多余的，可以被禁用。

缘由

Docker引擎提供了两种将主机端口转发到容器—hairpin NAT和用户级代理—机制。 在大多数情况下，hairpin NAT模式是首选，因为它提高了性能，并使用本地Linux iptables功能，而不是一个额外的组件。 在hairpin NAT可用的情况下，应在启动时禁用用户级代理，以减少安装的攻击面。

审计

```
ps -ef | grep dockerd
```

确保 `--userland-proxy` 参数设置为 `false` 。

修正

以如下方式运行Docker daemon：

```
dockerd --userland-proxy=false
```

影响

某些带有较旧Linux内核的系统可能无法支持 hairpin NAT，因此需要用户级代理服务。 此外，一些网络设置可能会受到删除用户级代理的影响。

默认值

默认情况下，启用了用户级代理。

参考文献

```
1. http://windsock.io/the-docker-proxy/
2. https://github.com/docker/docker/issues/14856
3. https://github.com/docker/docker/issues/22741
4.
https://docs.docker.com/engine/userguide/networking/default_network/binding/
```

2.19 加密overlay 网络不同节点上的容器之间交换的数据（Scored）

配置适用性

- **Level 1- Docker**

描述

加密overlay 网络不同节点上的容器之间交换的数据。

缘由

默认情况下，overlay 网络不同节点上的容器之间交换的数据不会被加密。 这可能会暴露容器节点之间的流量。

审计

运行以下命令并确保每个overlay 网络已被加密。

```
docker network ls --filter driver=overlay --quiet | xargs docker network
inspect --
format '{{.Name}} {{ .Options }}'
```

修正

使用 `--opt encrypted` 标志创建 overlay 网络。

影响

无

默认值

默认情况下， overlay 网络不同节点上的容器之间交换的数据在Docker swarm模式下不进行加密。

参考文献

```
1. https://docs.docker.com/engine/userguide/networking/overlay-security-
model/
2. https://github.com/docker/docker/issues/24253
```

2.20 如果需要，应用daemon-wide范围的自定义seccomp配置文件（Not Scored）

配置适用性

- **Level 2- Docker**

描述

如果需要，您可以选择在 daemon-wide 级别使用自定义的seccomp配置文件，用以覆盖Docker默认的seccomp配置文件。

缘由

大量的系统调用暴露于每个用户级进程，其中许多系统调用在整个生命周期中都未被使用。 大多数应用程序不需要所有的系统调用，因此可以通过减少可用的系统调用来获益。 减少的一组系统调用减少了暴露给应用程序的内核面，从而临时提升应用程序安全性。 您可以应用您自己自定义的seccomp配置文件，而不是Docker的默认seccomp配置文件。 或者，如果Docker的默认配置文件适合您的环境，则可以选择忽略此建议。

审计

运行以下命令并查看`“Security Options”`部分列出的seccomp配置文件。 如果它是默认的，那就意味着，应用Docker的默认seccomp配置文件。

```
docker info --format '{{ .SecurityOptions }}'
```

修正

默认情况下，应用Docker的默认seccomp配置文件。 如果这适用于您的环境，则不需要采取任何行动。 或者，如果您选择应用自己的seccomp配置文件，请在daemon 启动时使用`--seccomp-profile`标志，或将其放在daemon 运行时参数文件中。

```
dockerd --seccomp-profile </path/to/seccomp/profile>
```

影响

错误配置的seccomp配置文件可能会中断您的容器环境。 Docker默认的阻塞调用已经仔细审查。 这些解决了容器环境中的一些关键漏洞/问题（例如，内核密钥环调用）。 所以，在重写默认值时，您应该非常小心。

默认值

默认情况下，Docker使用seccomp配置文件。

参考文献

```
1. https://github.com/docker/docker/pull/26276
```

2.21 避免生产中的实验性特征（Scored）

配置适用性

- **Level 1- Docker**

描述

避免生产中的使用实验性功能。

缘由

实验现在是一个docker daemon运行时的标志，而不需单独构建。`– experimental` 作为运行时标志传递给docker daemon，激活实验性功能。 实验现在被认为是一个稳定的版本，但有一些功能可能没有经过测试，不能保证API的稳定性。

审计

运行以下命令并确保在`“Server"`部分中的`“Experimental”`属性设置为 `“false”`。

```
docker version --format '{{ .Server.Experimental }}'
```

修正

不要将`--experimental`作为运行时参数传递给docker daemon。

影响

无

默认值

默认情况下，docker daemon不会激活实验性功能。

参考文献

```
1. https://github.com/docker/docker/issues/26713
2. https://github.com/docker/docker/pull/27223
```

2.22 使用Docker的 secret 管理命令来管理Swarm集群中的seret（Not Scored）

配置适用性

- **Level 2- Docker**

描述

使用Docker的内置 `secret` 管理命令。

缘由

Docker有各种命令来管理Swarm集群中的secrets。 这是Docker 支持未来secret的基础，可能会有改进，如支持Windows，支持不同的存储等。

审计

如果可以，在swarm 管理节点上，运行以下命令，并确保在您的环境中使用了 `docker secret`：

```
docker secret ls
```

修正

按照 `docker secret` 文档，使用它来有效地管理 secret。

影响

无

默认值

不适用

参考文献

```
1. https://github.com/docker/docker/pull/27794
```

2.23 在自动锁定模式下运行swarm 管理节点（Scored）

配置适用性

- **Level 1- Docker**

描述

在自动锁定模式下运行Docker swarm 管理节点。

缘由

当Docker重新启动时，用于加密 swarm 节点之间通信的TLS密钥以及用于加密和解密磁盘上Raft日志的密钥都被加载到每个管理节点的内存中。 您应该保护相互的TLS加密密钥和用于加密和解密Raft日志的密钥。 这个保护可以通过使用 `–autolock` 标志来初始化swarm来启用。 启用 `--autolock` 后，当Docker重新启动时，您必须先使用Docker在swarm初始化时生成的加密密钥解锁swarm。

审计

运行下面的命令。 如果输出密钥，则意味着swarm已经用 `--autolock` 标志初始化了。 如果输出 `no unlock key is set`，则意味着swarm 未使用 `–autolock` 标志进行初始化，不符合此基线。

```
docker swarm unlock-key
```

修正

如果您正在初始化swarm，使用下面的命令。

```
docker swarm init --autolock
```

如果要在现有swarm管理节点上设置 `--autolock`，请使用以下命令。

```
docker swarm update --autolock
```

影响

在没有用户的手动干预输入解锁密钥，自动锁定模式下的swarm 不会重新启动恢复。 在某些部署中，这可能不利于可用性。

默认值

默认情况下，swarm 管理节点不在自动锁定模式运行。

参考文献

```
1.
https://github.com/mstanleyjones/docker.github.io/blob/af7dfdba8504f9b102fb3
1a78cd08a06c33a8975/engine/swarm/swarm_manager_locking.md
```

2.24 定期轮换swarm管理节点的自动锁定秘钥（Not Scored）

配置适用性

- **Level 1- Docker**

描述

定期轮换swarm 管理节点的自动锁定密钥。

缘由

Swarm管理节点的自动锁定秘钥不会自动轮换。 作为最佳做法，您应该定期轮换它们。

审计

目前，没有机制来找出密钥在swarm管理节点上最后一次轮换的时间。 如果有密钥轮转记录，并且按预定义的频率轮换密钥，则应该与系统管理员核对。

修正

运行下面的命令来轮换秘钥。

```
docker swarm unlock-key --rotate
```

此外，为了便于审计，维护密钥轮换记录并确保为密钥轮换建立预定义的频率。

影响

无

默认值

默认情况下，秘钥不会自动轮换。

参考文献

```
1.
https://github.com/mstanleyjones/docker.github.io/blob/af7dfdba8504f9b102fb3
1a78cd08a06c33a8975/engine/swarm/swarm_manager_locking.md
```

3 docker daemon 配置文件

本节涵盖了Docker相关文件和目录的权限和所有权。 保持可能包含敏感参数的文件和目录安全，对于Docker daemon 的正确和安全运行非常重要。

3.1 验证docker.service文件的所有权设置为root：root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`“docker.service”`文件所有权和组所有权已正确设置为`“root”`。

缘由

`“docker.service”`文件包含了一些可能会改变Docker daemon行为的敏感参数。因此，它应该由`“root”`用户和`“root”`用户组拥有，以此来保证文件的完整性。

审计

步骤1：找出文件位置

```
systemctl show -p FragmentPath docker.service
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，以验证该文件是否由`“root”`用户和`“root”`用户组拥有

例如：

```
stat -c %U:%G /usr/lib/systemd/system/docker.service | grep -v root:root
```

正常情况上述命令应该没有任何返回

修正

步骤1：找出文件位置：

```
systemctl show -p FragmentPath docker.service
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，将文件的用户和用户组设置为`“root”`

例如：

```
chown root:root /usr/lib/systemd/system/docker.service
```

影响

无

默认值

该文件可能不存在于系统上。在这种情况下，此基线不适用。默认情况下，如果文件存在，则该文件的用户和用户组应被正确的设置为`“root”`。

参考文献

```
1. https://docs.docker.com/engine/admin/systemd/
```

3.2 验证docker.service文件权限是否被设置为644或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`“docker.service”`文件的权限是否正确的设置为`“644”`或更严格

缘由

`“docker.service”`文件包含一些可能改变Docker daemon行为的敏感参数。因此除了`“root”`以外，其他任何用户的写入操作都是不被允许的，以此来保持文件的完整性。

审计

步骤1：找出文件位置

```
systemctl show -p FragmentPath docker.service
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，以验证文件权限是否被设置为`“644”`或更严格

例如，

```
stat -c %a /usr/lib/systemd/system/docker.service
```

修正

步骤1：找出文件位置

```
systemctl show -p FragmentPath docker.service
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，将文件权限设置为`“644”`

例如，

```
chmod 644 /usr/lib/systemd/system/docker.service
```

影响

无

默认值

该文件可能不存在于系统上。在这种情况下，此基线不适用。默认情况下，如果文件存在，则该文件权限应被正确设置为`“644”`。

参考文献

```
1. https://docs.docker.com/articles/systemd/
```

3.3 验证docker.socket文件的所有权设置为root：root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`“docker.socket”`文件的所有权和组所有权是否正确设置为`“root”`。

缘由

`“docker.socket”`文件包含一些可能会改变Docker远程API行为的敏感参数。因此，它应该由`“root”`用户和用户组所有，以此来保持文件的完整性。

审计

步骤1：找出文件位置：

```
systemctl show -p FragmentPath docker.socket
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，以验证该文件是否由`“root”`用户和`“root”`用户组拥有

例如：

```
stat -c %U:%G /usr/lib/systemd/system/docker.socket | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

步骤1：找出文件位置

```
systemctl show -p FragmentPath docker.socket
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令将文件的用户和用户组设置为`“root”`

例如，

```
chown root:root /usr/lib/systemd/system/docker.socket
```

影响

无

默认值

该文件可能不存在于系统上。在这种情况下，此基线不适用。默认情况下，如果文件存在，则该文件的用户和用户组应被正确的设置为`“root”`。

参考文献

```
1. https://docs.docker.com/articles/basics/#bind-docker-to-another-hostport-
or-a- unix-socket
2. https://github.com/YungSang/fedora-atomic-
packer/blob/master/oem/docker.socket
3. http://daviddaeschler.com/2014/12/14/centos-7rhel-7-and-docker-
containers- on-boot/
```

3.4 验证docker.socket文件权限是否被设置为644或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`“docker.socket”`文件的权限是否正确的设置为`“644”`或更严格

缘由

`“docker.socket”`文件包含一些可能改变Docker远程API行为的敏感参数。因此应该只有`“root”`用户才能对其进行写入操作，以此来保持文件的完整性

审计

步骤1：找出文件位置

```
systemctl show -p FragmentPath docker.socket
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，以验证文件权限是否被设置为`“644”`或更严格

例如，

```
stat -c %a /usr/lib/systemd/system/docker.socket
```

修正

步骤1：找出文件位置

```
systemctl show -p FragmentPath docker.socket
```

步骤2：如果文件不存在，则此基线不适用。如果该文件存在，请使用正确的文件路径执行以下命令，将文件权限设置为`“644”`

例如，

```
chmod 644 /usr/lib/systemd/system/docker.socket
```

影响

无

默认值

该文件可能不存在于系统上。在这种情况下，此基线不适用。默认情况下，如果文件存在，则该文件权限应被正确设置为`“644”`。

参考文献


```
1. https://docs.docker.com/articles/basics/#bind-docker-to-another-hostport-
or-a- unix-socket
2. https://github.com/YungSang/fedora-atomic-
packer/blob/master/oem/docker.socket
3. http://daviddaeschler.com/2014/12/14/centos-7rhel-7-and-docker-
containers- on-boot/
```

3.5 验证/etc/docker目录的所有权是否设置为root：root（Scored）

配置适用性

- Level 1- Docker

描述

验证`“/etc/docker“`目录的所有权和组所有权是否正确设置为`“root”`。

缘由

除了各种敏感文件，`“/etc/docker”`目录还包含证书和密钥。因此，它应该由`“root”`用户和用户组拥有，以此来保持目录的完整性。

审计

执行以下命令来验证该目录是否由`“root”`用户和用户组拥有

```
stat -c %U:%G /etc/docker | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root /etc/docker
```

这会将该目录的用户和用户组所有权设置为`“root”`

影响

无

默认值

默认情况下，此目录的用户和用户组所有权已正确设置为`“root”`

参考文献

```
1. https://docs.docker.com/articles/certificates/
```

3.6 验证/etc/docker目录权限是否被设置为755或更严格（Scored）

配置适用性

- Level 1- Docker

描述

验证`/etc/docker`目录权限是否被正确设置为`'755'`或更严格

缘由

除了各种敏感文件，`“/etc/docker”`目录还包含证书和密钥。因此只有`“root”`用户才能进行写入操作，以此来保持目录的完整性

审计

执行以下命令以验证目录权限是否为`“755”`或更严格

```
stat -c %a /etc/docker
```

修正

```
chmod 755 /etc/docker
```

这将设置目录的权限为`'755'`

影响

无

默认值

默认情况下，此目录的权限已正确设置为“755”

参考文献

```
1. https://docs.docker.com/articles/certificates/
```

3.7 验证仓库证书文件所有权是否设置为root：root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证所有仓库证书文件（通常在`/etc/docker/certs.d/<registry-name>`目录下）是否由`"root"`用户和用户组拥有

缘由

`/etc/docker/certs.d/ <registry-name>`目录包含Docker 仓库的证书。这些证书文件必须由`“root”`用户和用户组拥有，以维护证书的完整性

审计

执行以下命令以验证仓库证书文件是否由`"root"`用户和用户组拥有

```
stat -c %U:%G /etc/docker/certs.d/* | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root /etc/docker/certs.d/<registry-name>/*
```

这会将仓库证书文件的用户和用户组所有权设置为`“root”`

影响

无

默认值

默认情况下，仓库证书文件的用户和用户组所有权已正确设置为`“root”`

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/reference/commandline/cli/#insecure-registries
```

3.8 验证仓库证书文件权限是否设置为444或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证所有仓库证书文件（通常在`/etc/docker/certs.d/<registry-name>`目录下）的权限是否是`"444"`或更严格。

缘由

`/etc/docker/certs.d/<registry-name>`目录包含了Docker 仓库证书。这些证书文件必须具有`“444”`权限，以此维护证书的完整性

审计

执行以下命令以验证仓库证书文件权限为`“444”`或更严格

```
stat -c %a /etc/docker/certs.d/<registry-name>/*
```

修正

```
chmod 444 /etc/docker/certs.d/<registry-name>/*
```

这会将仓库证书文件的所有权设置为`“444”`

影响

无

默认值

默认情况下，仓库证书文件的权限可能不是`“444”`。默认文件权限由系统或用户特定的`umask`值进行管理

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/reference/commandline/cli/#insecure-registries
```

3.9 验证TLS CA证书文件的所有权是否设置为root：root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证TLS CA证书文件（与`“--tlscacert”`参数一起传递的文件）是否由`"root"`用户和用户组拥有

缘由

TLS CA证书文件应该被保护，以免受任何篡改。它通过给定的CA证书对Docker服务进行身份验证。因此，它必须由“`"root"`用户和用户组拥有，以保持CA证书的完整性

审计

执行以下命令以验证TLS CA证书文件是否由`"root"`用户和用户组拥有

```
stat -c %U:%G <path to TLS CA certificate file> | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root <path to TLS CA certificate file>
```

这会将TLS CA证书文件的用户和用户组所有权设置为`"root"`

影响

无

默认值

默认情况下，TLS CA证书文件的用户和用户组所有权已正确设置为`“root”`

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/articles/https/
```

3.10 验证TLS CA证书文件权限是否设置为444或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证TLS CA证书文件（与`“ -- tlscacert”`参数一起传递的文件）的权限是否是`"444"`或更严格。

缘由

TLS CA证书文件应该被保护，以免受任何篡改。它通过给定的CA证书对Docker服务进行身份验证。因此，它必须具有`“444”`权限，以保持CA证书的完整性

审计

执行以下命令以验证TLS CA文件权限为`“444”`或更严格

```
stat -c %a <path to TLS CA certificate file>
```

修正

```
chmod 444 <path to TLS CA certificate file>
```

这会将TLS CA文件的权限设置为`“444”`

影响

无

默认值

默认情况下，TLS CA文件的权限可能不是`“444”`。默认文件权限由系统或用户特定的`umask`值进行管理

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/articles/https/
```

3.11 验证Docker服务证书文件的所有权是否设置为root:root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证Docker服务证书文件（与`“--tlscert”`参数一起传递的文件）是否由`“root”`用户和用户组拥有

缘由

Docker服务证书文件应该被保护，以免受任何篡改。它通过给定的服务证书对Docker服务进行身份验证。因此，它必须由“`"root"`用户和用户组拥有，以保持证书的完整性

审计

执行以下命令以验证Docker服务证书文件是否由`"root"`用户和用户组拥有

```
stat -c %U:%G <path to Docker server certificate file> | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root <path to Docker server certificate file>
```

这会将Docker服务证书文件的用户和用户组所有权设置为`“root”`

影响

无

默认值

默认情况下，Docker服务证书文件的用户和用户组所有权已正确设置为`“root”`

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/articles/https/
```

3.12 验证Docker服务证书文件权限是否设置为444或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证Docker服务证书文件（与`“--tlscert”`参数一起传递的文件）的权限是否是`"444"`或更严格

缘由

Docker服务证书文件应该被保护，以免受任何篡改。它通过给定的服务证书对Docker服务进行身份验证。因此，它必须具有`“444”`权限，以保持Docker服务证书的完整性

审计

执行以下命令以验证Docker服务证书文件权限为`“444”`或更严格

```
stat -c %a <path to Docker server certificate file>
```

修正

```
chmod 444 <path to Docker server certificate file>
```

这会将Docker服务证书文件的权限设置为`“444”`

影响

无

默认值

默认情况下，Docker服务证书文件的权限可能不是`“444”`。默认文件权限由系统或用户特定的`umask`值进行管理

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/articles/https/
```

3.13 验证Docker服务证书密钥文件的所有权是否设置为root:root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证Docker服务证书密钥文件（与`“--tlskey”`参数一起传递的文件）是否由`“root”`用户和用户组拥有

缘由

Docker服务证书密钥文件应被保护，以防止被篡改以及任何不必要的读取。它拥有Docker服务证书的私钥。因此，它必须由`“root”`用户和用户组拥有，以保持证书的完整性

审计

执行以下命令以验证Docker服务证书密钥文件是否由`"root"`用户和用户组拥有

```
stat -c %U:%G <path to Docker server certificate key file> | grep -v
root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root <path to Docker server certificate key file>
```

这会将Docker服务证书密钥文件的用户和用户组所有权设置为`“root”`

影响

无

默认值

默认情况下，Docker服务证书密钥文件用户和用户组的所有权已正确设置为`“root”`

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/articles/https/
```

3.14 验证Docker服务证书密钥文件权限是否设置为400（Scored）

配置适用性

- **Level 1- Docker**

描述

验证Docker服务证书密钥文件（与`“--tlskey”`参数一起传递的文件）的权限是否是`"400"`

缘由

Docker服务证书密钥文件应被保护，以防止任何篡改及不必要的读取。它拥有Docker服务器证书的私钥。因此必须具有`“400”`的权限，以保持Docker服务证书的完整性

审计

执行以下命令来验证Docker服务证书密钥文件的权限是否为`“400”`

```
stat -c %a <path to Docker server certificate key file>
```

修正

```
chmod 400 <path to Docker server certificate key file>
```

这会将Docker服务证书密钥文件权限设置为`“400”`

影响

无

默认值

默认情况下，Docker服务证书密钥文件的权限可能不是`“400”`。默认文件权限由系统或用户特定的`umask`值进行管理

参考文献

```
1. https://docs.docker.com/articles/certificates/
2. http://docs.docker.com/articles/https/
```

3.15 验证Docker套接字文件所有权是否设置为root:docker（Scored）

配置适用性

- **Level 1- Docker**

描述

验证Docker套接字文件是由`“root”`用户拥有，且被`“docker”`用户组拥有

缘由

Docker daemon以`“root”`身份运行。因此，默认的Unix套接字必须由`“root”`用户拥有。如果任何其他用户或进程拥有此套接字，那么该非特权用户或进程就可能与Docker daemon 进行交互。而且，这样的非特权用户或进程还可能与容器交互。这既不安全也是我们不希望的行为。 此外，Docker安装程序创建一个名为`“docker”`的Unix组。您可以将用户添加到该组，这些用户将能够读写默认的Docker Unix套接字。`“docker”`组的成员资格由系统管理员严格控制。如果其他任何组拥有此套接字，那么该组的成员可能会与Docker daemon 进行交互。并且，这样的用户组可能不

像“docker”组那样受到严格控制。这既不安全也是我们不希望看到的行为。 因此，默认的Docker Unix套接字文件必须由“root”用户拥有，并由“docker”用户组拥有，以保持套接字文件的完整性

审计

执行以下命令验证Docker套接字文件是由“root”用户拥有，且由“docker”用户组拥有

```
stat -c %U:%G /var/run/docker.sock | grep -v root:docker
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:docker /var/run/docker.sock
```

这会将默认的Docker套接字文件的所有权和组所有权分别设置为“root”和“docker”

影响

无

默认值

默认情况下，Docker套接字文件的所有权和组所有权已被正确设置为“root：docker”

参考文献

```
1. https://docs.docker.com/reference/commandline/cli/#daemon-socket-option
2. https://docs.docker.com/articles/basics/#bind-docker-to-another-hostport-
or-a-
unix-socket
```

3.16 验证Docker套接字文件权限是否设置为660或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证Docker套接字文件权限是否为“660”或更严格

缘由

只允许“root”和“docker”组的成员读写默认的Docker Unix套接字。因此，Docket套接字文件权限是否为“660”或更严格

审计

执行以下命令以验证Docker套接字文件权限是否为“660”或更严格

```
stat -c %a /var/run/docker.sock
```

修正

```
chmod 660 /var/run/docker.sock
```

这会将Docker套接字文件的文件权限设置为“660”

影响

无

默认值

默认情况下，Docker套接字文件的权限已正确设置为“660”

参考文献

```
1. https://docs.docker.com/reference/commandline/cli/#daemon-socket-option
2. https://docs.docker.com/articles/basics/#bind-docker-to-another-hostport-
or-a-
unix-socket
```

3.17 验证daemon.json文件的所有权是否设置为root：root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证“daemon.json”文件是否由“root”用户和用户组拥有

缘由

"daemon.json"文件包含可能改变docker daemon行为的敏感参数。因此，它应该由“root”用户和用户组拥有，以保持文件的完整性

审计

执行以下命令以验证此文件是否由"root"用户和用户组拥有

```
stat -c %U:%G /etc/docker/daemon.json | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root /etc/docker/daemon.json
```

这会将此文件的用户和用户组所有权设置为`“root”`

影响

无

默认值

该文件可能不存在于系统上。 在这种情况下此基线不适用

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/#daemon-configuration-file
```

3.18 验证daemon.json文件权限是否被设置为644或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`“daemon.json”`文件权限是否正确设置为`“644”`或更严格

缘由

`"daemon.json"`文件包含可能改变docker daemon行为的敏感参数.因此，它应该由`“root”`用户和用户组拥有，以保持文件的完整性

审计

执行以下命令以验证此文件权限是否被正确设置为`“644”`或更严格

```
stat -c %a /etc/docker/daemon.json
```

修正

```
chmod 644 /etc/docker/daemon.json
```

这会将该文件的文件权限设置为`“644”`

影响

无

默认值

该文件可能不存在于系统上。 在这种情况下此基线不适用

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/daemon/#daemon-configuration-file
```

3.19 验证/etc/default/docker文件的所有权是否设置为root:root（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`“/etc/default/docker”`文件是否由`“root”`用户和用户组拥有

缘由

`”/etc/default/docker“`文件包含可能会改变docker daemon行为的敏感参数。因此，它应该由`“root”`用户和用户组拥有，以保持目录的完整性

审计

执行以下命令以验证此文件是否由`"root"`用户和用户组拥有

```
stat -c %U:%G /etc/default/docker | grep -v root:root
```

正常情况上述命令应该没有任何返回。

修正

```
chown root:root /etc/default/docker
```

这会将此文件的用户和用户组所有权设置为`“root”`

影响

无

默认值

该文件可能不存在于系统上。 在这种情况下此基线不适用

参考文献

```
1. https://docs.docker.com/engine/admin/configuring/
```

3.20 验证/etc/default/docker文件权限是否被设置为644或更严格（Scored）

配置适用性

- **Level 1- Docker**

描述

验证`/etc/default/docker`文件权限是否正确设置为`“644”`或更严格

缘由

`/etc/default/docker`文件包含可能改变docker daemon行为的敏感参数。因此，它应该由`“root”`用户和用户组拥有，以保持文件的完整性

审计

执行以下命令以验证此文件权限是否被正确设置为`“644”`或更严格

```
stat -c %a /etc/default/docker
```

修正

```
chmod 644 /etc/default/docker
```

这会将该文件的文件权限设置为`“644”`

影响

无

默认值

该文件可能不存在于系统上。 在这种情况下此基线不适用

参考文献

```
1. https://docs.docker.com/engine/admin/configuring/
```

4 容器镜像和构建文件

容器的基础镜像和构建文件管理着从一个特殊镜像启动容器的基本行为。确保您使用的是合适的基础镜像和恰当的构建文件，对于构建你的容器化基础服务非常重要。 为了确保您的容器化基础服务安全，您应该遵循以下关于容器基础镜像和构建文件的建议。

4.1 为容器创建一个用户（Scored）

配置适用性

- **Level 1- Docker**

描述

在容器镜像的Dockerfile中为容器创建一个非`root`用户。

缘由

如果可能，以非`root`用户运行容器是一个好习惯。 虽然现在可用使用用户名称空间映射，但如果已在容器映像中定义用户，则容器默认以该用户身份运行，并且不需要特定的用户名称空间重新映射。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: User={{
.Config.User }}'
```

上面的命令会返回容器的用户名和用户ID。如果返回空，那么意味着容器正在以root用户运行。

修正

确保容器镜像的Dockerfile包含以下指令:

```
USER <username or ID>
```

username or ID指的是可以在容器基础镜像中找到的用户。如果在容器基础镜像中没有创建特殊用户，那么在`USER`指令前添加一个`useradd`命令去添加一个特殊用户。

例如，在Dockerfile中添加下面的行，用于在容器中创建一个用户：

```
RUN useradd -d /home/username -m -s /bin/bash username
USER username
```

注意：如果在镜像中有容器不需要的用户，请考虑删除他们。删除这些用户后，提交镜像然后生成新的容器使用。

影响

无

默认值

默认情况下，容器以`root`权限运行，并且在容器内部使用`root`用户身份。

参考文献

```
1. https://github.com/docker/docker/issues/2918
2. https://github.com/docker/docker/pull/4572
3. https://github.com/docker/docker/issues/7906
4. https://www.altiscale.com/hadoop-blog/making-docker-work-yarn/
5. http://docs.docker.com/articles/security/
```

4.2 为容器使用可信的基础镜像（Not Scored）

配置适用性

- **Level 1- Docker**

描述

确保容器镜像是从头开始编写，或者是基于另一个通过安全通道下载的已制定的、可信的基础镜像。

缘由

官方仓库是由Docker社区或者供应商发布和优化的Docker镜像。可能还有其他或许不安全的公共仓库。因此在获取容器镜像时，您需要小心谨慎。

审计

通过执行下面的命令来检查Docker主机：

```
docker images
```

这将列出在当前Docker 主机上所有可用的容器镜像。咨询系统管理员，获取证明列表中的镜像都是通过安全通道从安全的源获取的证据

修正

配置和使用Docker信任的内容

影响

无

默认值

不适用

参考文献

1. https://titanous.com/posts/docker-insecurity
2. https://registry.hub.docker.com/
3. http://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-
security-options-mac-shared-directories/
4. https://github.com/docker/docker/issues/8093
5. http://docs.docker.com/reference/commandline/cli/#pull
6. https://github.com/docker/docker/pull/11109
7. https://blog.docker.com/2015/11/docker-trusted-registry-1-4/

4.3 不要在容器中安装不需要的包（Not Scored）

配置适用性

- **Level 1- Docker**

描述

容器往往是操作系统的最小精简版本。别安装任何不符合容器目的的东西。

缘由

用不必要的软件膨胀容器，可能会增加容器的受攻击面。这也违背了容器镜像最小精简的理念。因此，除了容器真正需要的，不要安装任何其他东西。

审计

步骤1：通过以下命令，列出所有运行的容器实例：

```
docker ps --quiet
```

步骤2：对于每个容器实例，执行以，或等价的命令：

```
docker exec $INSTANCE_ID rpm -qa
```

上面的命令会列出容器中的安装包。检查列表确保所有包的合理性。

修正

一开始，不要在容器上安装任何不合理的东西。如果镜像中有一些你的容器不需要的包，卸载他们。

如果可以，考虑使用最小的基础镜像而不是标准的`Redhat/Centos/Debian`镜像。例如，`BusyBox`和`Alpine`。

这不仅将你的镜像大小从150多M变为20M。还可以减少工具和路径提权途径。甚至你可以删除安装包，作为最终生产容器的加强措施。

影响

无

默认值

不适用

参考文献

```
1. https://docs.docker.com/userguide/dockerimages/
2. http://www.livewyer.com/blog/2015/02/24/slimming-down-your-docker-containers-alpine-linux
3. https://github.com/progrium/busybox
```

4.4 扫描、重新构建镜像，以包含安全补丁（Not Scored）

配置适用性

- **Level 1- Docker**

描述

镜像应该被经常扫描，以确保没有任何漏洞。重新构建包含安全补丁的镜像，然后实例化新容器。

缘由

漏洞是可以利用的bugs，安全补丁是为了消除漏洞而进行的升级。我们可以使用镜像漏洞扫描工具找到镜像中的各种漏洞，然后查找可用的安全补丁消除这些漏洞。补丁程序将系统更新到最新的代码库。 在当前的基础代码上是很重要的，因为这是厂商集中解决问题的地方。 在应用之前评估安全补丁并遵循补丁最佳实践。

另外，如果镜像漏洞扫描工具能够执行二进制级别分析或基于哈希的验证，而不是版本字符串匹配，效果会更好。

审计

步骤1：执行下面的命令，列出所有运行的容器：

```
docker ps --quiet
```

步骤2：为每一个容器实例，执行下面，或等价的命令，列出容器中的安装包。确保漏洞的安全补丁包已被安装。

```
 docker exec $INSTANCE_ID rpm -qa
```

或者，您可以在您的系统中运行能够扫描所有镜像的镜像漏洞扫描工具，然后为检测到的漏洞应用基于修复管理程序的补丁。

修正

按照以下步骤，使用安全补丁重新构建镜像：

步骤1：`'docker pull'`所有的基础镜像（也就是在Dockerfile集合中，提取在`'FROM'`指令中声明的所有镜像，并且重新拉取他们以检测更新/补丁版本）。也可在镜像中打补丁。

步骤2：使用`'docker build --no-cache'`强制重新构建每一个镜像。

步骤3：使用升级后的镜像重新开始所有的容器。

你也可以在Dockerfile中使用`ONBUILD`指令去引发镜像更新指令，常用于知道的会被经常用作基础镜像的镜像。

影响

无

默认值

默认情况下，容器和镜像是不会自己更新的。

参考文献

```
1. https://docs.docker.com/userguide/dockerimages/
2. https://docs.docker.com/docker-cloud/builds/image-scan/
3. https://blog.docker.com/2016/05/docker-security-scanning/
4. https://docs.docker.com/engine/reference/builder/#/onbuild
```

4.5 为Docker启用 Content trust（Scored)

配置适用性

- **Level 2- Docker**

描述

`Content trust`默认是不可用的，您应该启用它。

缘由

`Content trust`提供从远程Docker仓库发送和接受数据时的数字签名能力。这些签名允许客户端验证特定镜像标签的完整性和发布者。这可以确保容器镜像的来源。

审计

```
echo $DOCKER_CONTENT_TRUST
```

这应该返回1。

修正

在bash shell中启用`content trust`,输入下面的命令：

```
export DOCKER_CONTENT_TRUST=1
```

或者，在您的profile文件中设置环境变量，这样每次登录都会启用`content trust`。

影响

在一个设置了`DOCKER_CONTENT_TRUST`的环境中，在以`-buile,create,pull,pus和run`操作镜像时，你需要遵循信任流程。您可以使用`--disable-content-trust`标志在没有`content trust`标签的镜像上运行单独的操作，但是这违背了启用`content trust`的目的，因此应尽可能避免。

注意：`Content trust`当前只能在Docker Hub 的用户上使用。在Docker Trusted Registry或者私有仓库上还不可用。

默认值

默认，没有启用`content trust`。

参考文献

```
1. https://docs.docker.com/engine/security/trust/content_trust/
2. https://docs.docker.com/engine/reference/commandline/cli/#notary
3. https://docs.docker.com/engine/reference/commandline/cli/#environment-
variables
```

4.6 为容器镜像添加HEALTHCHECK指令（Scored）

配置适用性

- **Level 1- Docker**

描述

在您的docker容器镜像中添加`HEALTHCHECK`指令，完对运行中容器的健康检查。

缘由

安全三要素中，其中一个重要的是可用性。 将`HEALTHCHECK`指令添加到您的容器镜像，确保docker引擎定期检查正在运行的针对该指令的容器实例仍在工作。

基于健康状态记录，docker引擎可以让停止工作的容器退出，并且实例化一个新的。

审计

运行以下命令，确保docker镜像已经设置合适的`HEALTHCHECK`指令。

```
docker inspect --format='{{ .Config.Healthcheck }}' <IMAGE>
```

修正

按照Docker文档，使用`HEALTHCHECK`指令重新构建您的容器镜像。

影响

无

默认值

默认情况下，没有设置`HEALTHCHECK`。

参考文献

```
1. https://github.com/docker/docker/pull/22719
```

4.7 不要在Dockerfile中单独使用升级指令（Not Scored）

配置适用性

- **Level 1- Docker**

描述

在 Dockerfile中不要在一行、或者单独的使用更新指令，例如，`apt-get update` 。

缘由

在Dockerfile中使用单行的更新指令将会缓存更新层。因此，以后使用相同的指令构建任何镜像时，先前的缓存更新层将被使用。 这可能会在以后的版本中拒绝新的更新。

审计

步骤1：运行以下的命令，获取镜像列表：

```
docker images
```

步骤2：对上面的列表中每个镜像运行下面的命令，并查找任何在单行中的更新指令:

```
docker history <Image_ID>
```

或者，如果您可以访问该镜像的Dockerfile，请验证是否有如上描述的更新指令。

修正

在安装时，使用更新指令，后面带上安装说明（或任何其他）和固定版本。 这将破坏缓存并强制提取需要的版本。 或者，您可以在docker构建过程中使用`--no-cache`标志来避免使用缓存层。

影响

无

默认值

默认情况下，docker不会强制限制使用更新指令。

参考文献

```
1. https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#/apt-get
2. https://github.com/docker/docker/issues/3313
```

4.8 删除镜像中的setuid和setgid权限（Not Scored）

配置适用性

- **Level 2- Docker**

描述

删除镜像中的`setuid`和`setgid`权限可以预防容器中的提权攻击。

缘由

`setuid`和`setgid`权限可以用于提升权限。虽然这些权限有时是合法需求，但是这些权限可能会被提权攻击利用。因此，你应该考虑在不需要权限的镜像中删除这些包的权限。

审计

在镜像上运行以下命令，列出所有具有`setuid`和`setgid`权限的可执行文件。

```
docker run <Image_ID> find / -perm +6000 -type f -exec ls -ld {} \; 2> /dev/null
```

仔细查看列表，确保它都是合法的。

修正

仅在需要它们的可执行文件上允许`setuid`和`setgid`权限。 您可以通过在您的Dockerfile中加入下面的命令，最好靠近Dockerfile的末尾，以在编译时删除这些权限，：

```
RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true
```

影响

以上命令会破坏所有依赖`setuid`或`setgid`权限的可执行文件,包括合法的。 因此，要小心修改命令以适合您的要求，以便不会丢失合法程序的权限。 这个需要仔细检查每个可执行文件并对权限进行微调。

默认值

不适用

参考文献

```
1. http://www.oreilly.com/webops-perf/free/files/docker-security.pdf
2. http://container-
solutions.com/content/uploads/2015/06/15.06.15_DockerCheatSheet_A2.pdf
3. http://man7.org/linux/man-pages/man2/setuid.2.html
4. http://man7.org/linux/man-pages/man2/setgid.2.html
```

4.9 在Dockerfile中使用COPY代替ADD（Not Scored）

配置适用性

- **Level 1- Docker**

描述

在Dockerfile中，使用 `COPY` 指令代替 `ADD` 指令。

缘由

`COPY` 指令只是将本地主机上的文件复制到容器文件系统中。 `ADD` 指令可能会从远程URL检索文件并执行操作，如拆包。 因此， `ADD` 指令会引入风险，诸如从URL添加没有扫描和解包分析处理的恶意文件。

审计

步骤1：运行以下命令，获取镜像列表

```
docker images
```

步骤2：对上面列表的每一个镜像运行下面的命令，来查找 `ADD` 指令

```
docker history <Image_ID>
```

或者，如果您可以访问该映像的Dockerfile，请验证没有 `ADD` 指令。

修正

在Dockerfiles中使用 `COPY` 指令。

影响

你需要注意 `ADD` 指令提供的功能，例如从远程URL获取文件。

默认值

不适用

参考文献

```
1. https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#/add-or-copy
```

4.10 在Dockerfile中不要存储机密（Not Scored）

配置适用性

- **Level 1- Docker**

描述

在Dockerfiles中不要储存任何秘密。

缘由

通过使用本地Docker命令，例如，docker history，各种工具和实用程序等可以轻松回溯Dockerfiles。 此外，作为一种普遍的做法，镜像发布者提供Dockerfiles来构建镜像的可信度。 因此，这些Dockerfiles中的秘密可能很容易被暴露，并可能被利用。

审计

步骤1：运行下面的命令，获取镜像列表

```
docker images
```

步骤2：对上面列表中的每个镜像运行以下命令，查找可能的秘密：

```
docker history <Image_ID>
```

或者，如果你可以访问镜像的Dockerfile,请确认没有如上所述的秘密。

修正

不要在Dockerfiles中存储任何类型的秘密。

影响

您需要找到一种方法来处理Docker镜像的秘密。

默认值

默认情况下，在Dockerfiles中不限制存储配置机密。

参考文献

```
1. https://github.com/docker/docker/issues/13490
2. http://12factor.net/config
3. https://avicoder.me/2016/07/22/Twitter-Vine-Source-code-dump/
```

4.11只安装已验证的包（Not Scored）

配置适用性

- **Level 2- Docker**

描述

在镜像中安装包之前，证实它的可靠性。

缘由

验证软件包的真实性对于构建安全的容器镜像至关重要。篡改过的软件包可能是恶意的，或者有一些已知的漏洞可以被利用。

审计

步骤1：运行下面的命令，获取镜像列表

```
docker images
```

步骤2：对上面列表中的每个镜像运行以下命令，并寻找方法确定安装包的真实性。 可以通过使用GPG密钥或其他安全包分发机制：

```
docker history <Image_ID>
```

或者，如果您可以访问该镜像的Dockerfile，请验证被检查包的可靠性。

修正

使用GPG密钥下载和验证软件包，或者其他你选择的安全包分配机制。

影响

无

默认值

不适用

参考文献

```
1. http://www.oreilly.com/webops-perf/free/files/docker-security.pdf
2. https://github.com/docker-
library/httpd/blob/12bf8c8883340c98b3988a7bade8ef2d0d6dcf8a/2.4/Dockerfile
3. https://github.com/docker-
library/php/blob/d8a4ccf4d620ec866d5b42335b699742df08c5f0/7.0/alpine/Dockerf
ile
4. https://access.redhat.com/security/team/key
```

5 容器运行时

容器的启动方式对安全性有很大的影响。 可能提供潜在危险的运行时参数，这些参数可能会危及主机和主机上的其他容器。 验证容器运行时非常重要。 评估容器运行时的各种基线如下：

5.1 不要禁用AppArmor配置文件（Scored）- 此条有变化

配置适用性

- **Level 1- Docker**

描述

AppArmor是一个有效且易于使用的Linux应用程序安全系统。 它在很多Linux发行版上如Debian和Ubuntu都默认可用。

缘由

AppArmor通过执行被称为AppArmor配置文件的安全策略来保护Linux操作系统和应用程序免受各种威胁。 您可以为容器创建自己的AppArmor配置文件，也可以使用Docker的默认AppArmor配置文件。 这将在配置文件中定义的容器上执行安全策略。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
AppArmorProfile={{.AppArmorProfile }}'
```

上述命令应该为每个容器实例返回一个有效的AppArmor配置文件。

修正

如果您的Linux操作系统可以使用AppArmor，请使用它。 您可能需要按照以下步骤进行操作：

1. 验证AppArmor是否已安装。 如果没有，请安装它。
2. 为Docker容器创建或导入AppArmor配置文件。
3. 将此配置文件置于强制模式。
4. 使用定制的AppArmor配置文件启动您的Docker容器。 例如，

```
docker run --interactive --tty --security-opt="apparmor:PROFILENAME" centos
/bin/bash
```

或者，您可以保留doker的默认apparmor配置文件

影响

容器（进程）将具有AppArmor配置文件中定义的一组限制。 如果您的AppArmor配置文件配置错误，则容器可能无法按预期完成工作。

默认值

默认情况下，将使用位于`/etc/apparmor.d/docker`的`docker-default` AppArmor配置文件运行容器。

参考文献

```
1. https://docs.docker.com/engine/security/apparmor/
2. http://docs.docker.com/articles/security/#other-kernel-security-features
3. https://github.com/docker/docker/blob/master/docs/security/apparmor.md
4. http://docs.docker.com/reference/run/#security-configuration
```

5.2 如果适用，验证SELinux安全选项（Scored）

配置适用性

- **Level 2- Docker**

描述

SELinux是一个高效且易于使用的Linux应用程序安全系统。 它在默认情况下可用于很多Linux发行版，例如Red Hat和Fedora。

缘由

SELinux提供了强制访问控制（MAC）系统，大大增强了默认的自主访问控制（DAC）模型。 如果适用，您可以在Linux主机上启用SELinux，从而增加额外的安全性。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
SecurityOpt={{.HostConfig.SecurityOpt }}'
```

上述命令应该返回为当前容器配置的所有安全选项。

修正

如果SElinux适用于您的Linux操作系统，请按照以下步骤设：

1. 设置SElinux状态
2. 设置 SElinux策略
3. 为docker容器创建或者导入一个SElinux策略模板
4. 启用SElinux选项，启动docker daemon，例如，

   ```
   docker daemon --selinux-enabled
   ```

5. 使用安全选项启动docker容器，例如，

```
docker run --interactive --tty --security-opt label=level:TopSecret
centos
/bin/bash
```

影响

容器（进程）将具有SELinux策略中定义的一组限制。 如果您的SELinux策略配置错误，那么容器可能无法按预期完全工作。

默认值

默认情况下，不在容器上应用SELinux安全选项。

参考文献

```
1. http://docs.docker.com/articles/security/#other-kernel-security-features
2. http://docs.docker.com/reference/run/#security-configuration
3. http://docs.fedoraproject.org/en-US/Fedora/13/html/Security-
Enhanced_Linux/
```

5.3 限制容器内的Linux内核功能（Scored）

配置适用性

- **Level 1- Docker**

描述

默认情况下，Docker将启动一组受限制的Linux内核功能。 这意味着任何进程都可能被授予所需的功能，而不是`root`访问权限。 使用Linux内核功能，进程不必以超级用户身份运行几乎所有需要root权限的特定区域。

缘由

Docker支持添加和删除功能，允许使用非默认配置文件。 这可能会使Docker通过删除功能更安全，或者通过增加功能来降低Docker的安全性。 因此，建议删除除您的容器过程明确要求的所有功能。例如，容器过程通常不需要下面的功能：

```
NET_ADMIN
SYS_ADMIN
SYS_MODULE
```

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: CapAdd=
{{.HostConfig.CapAdd }} CapDrop={{ .HostConfig.CapDrop }}'
```

验证添加和删除的Linux内核功能是否与每个容器实例的容器进程所需的功能一致。

修正

执行以下命令添加所需的功能：

```
$> docker run --cap-add={"Capability 1","Capability 2"} <Run arguments>
<Container Image Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --cap-add={"NET_ADMIN","SYS_ADMIN"}
centos:latest
/bin/bash
```

执行以下命令删除不需要的功能：

```
$> docker run --cap-drop={"Capability 1","Capability 2"} <Run arguments>
<Container Image Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --cap-drop={"SETUID","SETGID"} centos:latest
/bin/bash
```

或者，您可以选择删除所有功能，只添加需要的功能：

```
$> docker run --cap-drop=all --cap-add={"Capability 1","Capability 2"} <Run
arguments> <Container Image Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --cap-drop=all --cap-add=
{"NET_ADMIN","SYS_ADMIN"}
centos:latest /bin/bash
```

影响

根据添加或删除哪些Linux内核功能，将适用于容器内的限制。

默认值

默认情况下，以下功能可用于容器：

```
AUDIT_WRITE
CHOWN
DAC_OVERRIDE
FOWNER
FSETID
```

```
KILL
MKNOD
NET_BIND_SERVICE
NET_RAW
SETFCAP
SETGID
SETPCAP
SETUID
SYS_CHROOT
```

参考文献


```
1. https://docs.docker.com/articles/security/#linux-kernel-capabilities
2.
https://github.com/docker/docker/blob/master/daemon/execdriver/native/temp
late/default_template.go
3. http://man7.org/linux/man-pages/man7/capabilities.7.html
4. http://www.oreilly.com/webops-perf/free/files/docker-security.pdf
```


5.4 不要使用特权容器（Scored）

配置适用性

- **Level 1- Docker**

描述

使用`--privileged`标志会将所有Linux内核功能赋予容器，从而覆盖`--cap-add`和`--cap-drop`标志。 确保它不被使用。

缘由

`--privileged`标志赋予容器所有的功能，同时也提升了设备`cgroup`控制器强制执行的所有限制。 换句话说，容器可以做主机可以做的几乎所有事情。 这个标志存在允许特殊的用例，比如在Docker中运行Docker。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
Privileged={{.HostConfig.Privileged }}'
```

以上命令应该为每个容器实例返回Privileged = false。

修正

不要使用`--privileged`标志运行容器。

例如，不要以如下方式启动一个容器：

```
docker run --interactive --tty --privileged centos /bin/bash
```

影响

除了默认值之外，Linux内核功能将不能在容器内使用。

默认值

`False`

参考文献

```
1. https://docs.docker.com/reference/commandline/cli
```

5.5 不要在容器上挂载敏感的主机系统目录（Scored）

配置适用性

- **Level 1- Docker**

描述

敏感的主机系统目录（如下面）不应该被允许作为容器卷挂载，特别是在读写模式下。

```
/
/boot
/dev
/etc
/lib
/proc
/sys
/usr
```

缘由

如果敏感目录以读写模式挂载，则可以对敏感目录中的文件进行更改。 这些更改可能会降低安全隐患，或可能导致Docker主机处于无法保证的受损状态。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
Volumes={{ .Mounts}}'
```

上述命令将返回当前映射目录的列表以及是否以读写模式为每个容器实例挂载。

修正

不要将主机敏感目录挂载到在容器上，特别是在读写模式下。

影响

无

默认值

Docker默认为读写卷，但也可以只读目录。 默认情况下，容器上不会安装敏感的主机目录。

参考文献

```
1. https://docs.docker.com/userguide/dockervolumes
```

5.6 不要在容器中运行ssh（Scored）

配置适用性

- **Level 1- Docker**

描述

SSH服务不应该在容器内运行。 您应该SSH进入Docker主机，并使用nsenter工具从远程主机进入容器。

缘由

在容器中运行SSH会增加安全管理的复杂性：

- 难以管理SSH服务的访问策略和安全合规性
- 很难管理各个容器中的密钥和密码
- 难以管理SSH服务的安全升级

可以在没有使用SSH的情况下使用shell访问容器，应该避免不必要地增加安全管理的复杂性。

审计

步骤1：通过执行以下命令列出所有正在运行的容器实例：

```
docker ps --quiet
```

步骤2：对于每个容器实例，执行以下命令：

```
docker exec $INSTANCE_ID ps -el
```

确保没有SSH服务进程。

修正

从容器中卸载SSH服务，并使用nsenter或任何其他命令（如docker exec或docker attach）与容器实例进行交互。

```
docker exec --interactive --tty $INSTANCE_ID sh
```

或者

```
docker attach $INSTANCE_ID
```

影响

无

默认值

默认情况下，SSH服务不在容器中运行。 每个容器只允许一个进程。

参考文献

```
1. http://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker/
```

5.7 不要给容器映射特权端口（Scored）

配置适用性

- **Level 1- Docker**

描述

低于1024的TCP / IP端口号被认为是特权端口。 普通用户和进程不允许出于各种安全原因使用它们。 Docker允许一个容器端口映射到一个特权端口。

缘由

默认情况下，如果用户没有明确声明容器端口为主机端口映射，则Docker将自动并正确地将容器端口映射到主机上的49153- 65535块中可用端口。 但是，如果用户显式声明，则允许将容器端口映射到主机上的特权端口。 这是因为容器是使用NET_BIND_SERVICE Linux内核功能执行的，并不限制特权端口映射。 特权端口接收和传输各种敏感和特权数据。 允许容器使用它们会带来严重的影响。

审计

通过执行以下命令列出所有正在运行的容器实例及其端口映射：

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: Ports=
{{.NetworkSettings.Ports }}'
```

查看列表并确保容器端口未映射到低于1024的主机端口号。

修正

启动容器时不要将容器端口映射到主机特权端口。 另外，确保在Dockerfile中没有托管特权端口映射的声明。

影响

无

默认值

默认情况下，允许将一个容器的端口映射到主机上的一个特权端口。

注意：在某些情况下，您想要映射特权端口，因为如果禁止它，则相应的应用程序必须在容器之外运行。 例如：HTTP和HTTPS负载均衡器必须分别绑定80 / tcp和443 / tcp。 禁止映射特权端口有效地禁止在容器中运行这些端口，并且要求使用外部负载平衡器。 在这种情况下，这些容器实例应该被标记为这个建议的例外。

参考文献

```
1. http://docs.docker.com/articles/networking/#binding-ports
2. https://www.adayinthelifeof.nl/2012/03/12/why-putting-ssh-on-another-
port-
than-22-is-bad-idea
```

5.8 打开容器上只需要的端口（Scored）

配置适用性

- **Level 1- Docker**

描述

容器镜像的Dockerfile定义了容器实例上默认打开的端口。 端口列表可能与您在容器中运行的应用程序相关，也可能不相关。

缘由

一个容器可以运行Dockerfile中为其镜像定义的端口，或者可以任意传递运行时参数来打开一个端口列表。 此外，Dockerfile可能会经历各种更改，并且暴露的端口列表可能与您在容器中运行的应用程序相关，也可能不相关。 打开不需要的端口增加了容器、容器化应用程序的攻击面。 作为推荐的做法，不要打开不需要的端口。

审计

通过执行以下命令列出所有正在运行的容器实例及其端口映射：

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: Ports=
{{.NetworkSettings.Ports }}'
```

查看列表并确保映射的端口是容器真正需要的端口。

修正

修复容器镜像的Dockerfile，以便仅由您的容器化应用程序公开所需的端口。 你也可以通过在启动容器时不使用`'-P'`（UPPERCASE）或`'--publish-all'`标志来完全忽略在Dockerfile中定义的端口列表。 使用`'-p'`（小写）或`'--publish'`标志来明确定义特定容器实例所需的端口。 例如，

```
docker run --interactive --tty --publish 5000 --publish 5001 --publish 5002
centos
/bin/bash
```

影响

无

默认值

默认情况下，在使用`“-P”`或`“--publish-all”`标志运行容器时，会打开在镜像Dockerfile中`EXPOSE`指令下的列出的所有端口。

参考文献

```
1. https://docs.docker.com/articles/networking/#binding-ports
```

5.9 不要共享主机的网络名称空间（Scored）

配置适用性

- **Level 1- Docker**

描述

设置为`“--net = host”`时，容器上的联网模式将跳过将容器放置在单独的网络堆栈中。 实质上，这个选择告诉Docker不要容器化容器网络。 这将在网络方面意味着容器处于Docker主机的“外部”，并完全访问其网络接口。

缘由

这是潜在的危险。 它允许容器进程像任何其他`root`进程一样打开低编号的端口。 它还允许容器访问Docker主机上的D-bus 等网络服务。 因此，容器进程可能会做出意想不到的事情，比如关闭Docker主机。 你不应该使用这个选项。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
NetworkMode={{.HostConfig.NetworkMode }}'
```

如果上述命令返回`“NetworkMode = host”`，则意味着在启动容器时传递了`“--net = host”`选项。 这是不符合基线的。 它应该返回`bridge`，`none`，或`container:$Container_Instance`。

修正

启动容器时不要传递`'--net = host'`选项。

影响

无

默认值

默认情况下，容器连接到Docker `bridge`。

参考文献

```
1. http://docs.docker.com/articles/networking/#how-docker-networks-a-container
2. https://github.com/docker/docker/issues/6401
```

5.10 限制容器的内存使用量（Scored）

配置适用性

- **Level 1- Docker**

描述

默认情况下，Docker主机上的所有容器均等共享资源。 通过使用Docker主机的资源管理功能，例如内存限制，您可以控制容器可能消耗的内存量。

缘由

默认情况下，容器可以使用主机上的所有内存。 您可以使用内存限制机制来防止由于一个容器消耗了所有主机资源而导致的拒绝服务，以致同一主机上的其他容器无法执行其预期的功能。 对内存没有限制可能导致一个容器容易使整个系统不稳定并因此不可用的问题。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: Memory={{.HostConfig.Memory }}'
```

如果上述命令返回0，则意味着内存限制不到位。 如果上面的命令返回一个非零值，这意味着内存限制已经到位。

修正

根据需要只用尽可能多的内存来运行容器。 始终使用`'–memory'`参数运行容器。 你应该以如下方式启动容器：

```
$> docker run <Run arguments> --memory <memory-size> <Container Image Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --memory 256m centos /bin/bash
```

在上面的示例中，容器启动的内存限制为256 MB。 **注意：请注意，如果内存限制到位，下面命令的输出将以科学记数法返回值。**

```
docker inspect --format='{{.Config.Memory}}' 7c5a2d4c7fe0
```

上面的`7c5a2d4c7fe0`为容器ID。

例如，如果上述容器实例的内存限制设置为256 MB，则上述命令的输出将是2.68435456e + 08而不是256m。 您应该使用科学计算器或编程方法来转换此值。

影响

如果你没有设置适当的限制，容器过程可能不得不饿死。

默认值

默认情况下，Docker主机上的所有容器均等共享资源。 没有强制执行内存限制。

参考文献

1. https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/
2. http://docs.docker.com/reference/commandline/cli/#run
3. https://docs.docker.com/articles/runmetrics/

5.11 适当设置容器CPU的优先级（Scored）

配置适用性

- **Level 1- Docker**

描述

默认情况下，Docker主机上的所有容器均等共享资源。 通过使用Docker主机的资源管理功能，例如CPU共享，可以控制容器可能使用的主机CPU资源。

缘由

默认情况下，CPU时间在容器间平均分配。 如果需要，为了控制容器实例中的CPU时间，可以使用CPU共享功能。 CPU共享允许将一个容器优先于另一个容器，并且禁止较低优先级的容器更频繁地声明CPU资源。 这确保高优先级的容器更好地服务。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
CpuShares={{.HostConfig.CpuShares }}'
```

如果上述命令返回0或1024，则意味着CPU份额不到位。 如果上述命令返回非1024、非零值，则意味着CPU份额已到位。

修正

管理您容器之间的CPU份额。 为此，请使用`'--cpu-shares'`参数启动容器。 您可以如下方式启动容器：

```
$> docker run <Run arguments> --cpu-shares <CPU shares> <Container Image Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --cpu-shares 512 centos /bin/bash
```

在上面的例子中，容器以其他容器使用的50%的CPU份额启动。 所以，如果另一个容器具有80%的CPU份额，则该容器将具有40%的CPU份额。

注意：默认情况下，每个新的容器将有1024个CPU份额。 但是，如果运行审计部分中提到的命令，则此值显示为“0”。

或者：

1. 导航到`/sys/fs/cgroup/cpu/system.slice/`目录。
2. 使用`“docker ps”`命令检查您的容器实例ID。
3. 现在，在上面的目录中（在步骤1），你将有一个目录的名称为：

   `'docker- <Instance ID> .scope'`例如`'docker-4acae729e8659c6be696ee35b2237cc1fe4edd2672e9186434c5116e1a6fbed6.scope'`。 导航到这个目录。

   最新版没有此目录，直接下一步。
4. 您将找到一个名为`“cpu.shares”`的文件。 执行`'cat cpu.shares'`。 这将始终为您提供基于系统的CPU份额值。 因此，即使在`“docker run”`命令中没有使用`“-c”`或`“--cpu-shares”`参数配置的CPU份额，该文件的值也将为`“1024”`。

如果我们将一个容器的CPU份额设置为512，则与其他容器相比，它将获得一半的CPU时间。 所以，以1024为100%，然后做快速的数学推导出你应该为各自的CPU份额设置的数字。 例如，如果要设置50%，则使用512，如果要设置25%，则使用256。

影响

如果您没有设置正确的CPU份额，那么如果主机上的资源不可用，则容器进程可能不得不饿死。 如果主机上的CPU资源空闲，则CPU份额不会对容器可能使用的CPU进行任何限制。

默认值

默认情况下，Docker主机上的所有容器均等共享资源。 没有执行CPU份额。

参考文献

```
1. https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/
2. http://docs.docker.com/reference/commandline/cli/#run
3. https://docs.docker.com/articles/runmetrics/
```

5.12 将容器的根文件系统挂载为只读（Scored）

配置适用性

- **Level 1- Docker**

描述

容器的根文件系统应该被视为“黄金镜像”，并且应该避免对根文件系统的任何写入。 你应该明确地定义一个容器的卷来写入。

缘由

你不应该在容器中写数据。 应该明确定义和管理属于容器的数据卷。 这在管理员希望控制开发人员写文件和错误的情况下很有用。 另外，这还有其他的好处，例如：

- 这基础结构不会变化
- 由于无法写入容器实例，因此不需要审计实例差异
- 因为实例不能被篡改或写入，减少安全攻击媒介
- 能够使用基于卷的备份，而无需备份任何实例

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
ReadonlyRootfs={{.HostConfig.ReadonlyRootfs }}'
```

如果上述命令返回`“true”`，则意味着根文件系统是只读的。 如果上述命令返回`“false”`，则表示根文件系统是可写的。

此外，您可以使用下面的命令来查找容器实例与其相应镜像之间的差异。

```
docker diff $INSTANCE_ID
```

修正

添加一个`“--read-only”`标志，允许将容器的根文件系统作为只读进行挂载。 这可以与卷组合使用，以强制容器的进程只写入将被持久化的位置。 您应该以如下方式运行容器：

```
$> docker run <Run arguments> --read-only -v <writable-volume> <Container
Image Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --read-only --volume /centdata centos
/bin/bash
```

这将使用只读的根文件系统运行容器，并使用`“centdata”`作为容器卷进行写入。

影响

容器根文件系统将不可写入。 您应该明确地为容器定义一个卷来写入。

默认值

默认情况下，容器的根文件系统是可写的，允许进程在任何地方写文件。

参考文献

```
1. http://docs.docker.com/reference/commandline/cli/#run
```

5.13 将传入的容器流量绑定到特定的主机接口（Scored）

配置适用性

- **Level 1- Docker**

描述

默认情况下，Docker容器可以连接到外部世界，但是外部世界不能连接到容器。 每个传出连接似乎都来自主机自己的一个IP地址。 只允许通过主机上的特定外部接口联系容器服务。

缘由

如果主机上有多个网络接口，则容器可以接受任何网络接口上暴露端口的连接。 这可能是不希望的，可能不安全。 很多时候，一个特定的接口暴露在外部，在这些接口上运行诸如入侵检测，入侵防御，防火墙，负载平衡等服务来屏蔽传入的公共流量。 因此，你不应该接受任何接口上的传入连接。 您应该只允许来自特定外部接口的传入连接。

审计

通过执行以下命令列出所有正在运行的容器实例及其端口映射：

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: Ports=
{{.NetworkSettings.Ports }}'
```

查看列表并确保公开的容器端口绑定到特定的接口，而不是通配符IP地址 - `“0.0.0.0”`。

例如， 如果上面的命令返回如下，那么容器可以接受指定端口49153上的任何主机接口上的连接，这是不符合基线的。

```
Ports=map[443/tcp:<nil> 80/tcp:[map[HostPort:49153 HostIp:0.0.0.0]]]
```

但是，如果将暴露的端口连接到主机上的特定接口（如下所示），则此基线将根据需要进行配置并符合规定。

```
Ports=map[443/tcp:<nil> 80/tcp:[map[HostIp:10.2.3.4 HostPort:49153]]]
```

修正

将容器端口绑定到所需主机端口上的特定主机接口。 例如，

```
docker run --detach --publish 10.2.3.4:49153:80 nginx
```

在上面的示例中，容器端口80绑定到49153上的主机端口，并且只接受来自10.2.3.4外部接口的入站连接。

影响

无

默认值

默认情况下，Docker公开的容器端口为`0.0.0.0`，该通配符IP地址将与主机上任何可能的入网网络接口相匹配。

参考文献

```
1. https://docs.docker.com/articles/networking/#binding-container-ports-to-the-host
```

5.14 将'on-failure'容器重启策略设置为5（Scored）

配置适用性

- **Level 1- Docker**

描述

在`“docker run”`命令中使用`“--restart”`标志，您可以指定一个重启策略，以便在退出时应该或不应该重启容器。 您应该选择`“on- failure”`重新启动策略，并将重新启动尝试次数限制为5次。

缘由

如果无限期地继续尝试启动容器，可能会导致主机上的拒绝服务。 这可能是一个简单的方法来做分布式拒绝服务攻击，特别是如果你在同一个主机上有很多容器。 此外，忽略容器的退出状态和`“always"`尝试重新启动容器会导致无法调查容器被终止的根本原因。 如果一个容器被终止，您应该调查它的原因，而不是试图无限期地重启它。 因此，建议使用`“on-failure”`重新启动策略，并将其限制为最多5次重新启动尝试。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id
}}:RestartPolicyName={{ .HostConfig.RestartPolicy.Name }}
MaximumRetryCount={{.HostConfig.RestartPolicy.MaximumRetryCount }}'
```

- 如果上面的命令返回`'RestartPolicyName = always'`，那么系统没有按照需要进行配置，因此是不符合基线的。
- 如果以上命令返回`“RestartPolicyName = no”`或`“RestartPolicyName =”`，则重新启动策略将不会被使用，并且容器将永远不会被重新启动。 这个是不适用的，可以被认为是合规的。
- 如果上述命令返回`“RestartPolicyName = on-failure”`，则通过查看`“MaximumRetryCount”`来验证重新启动尝试的次数是否设置为5或更少。

修正

如果一个容器需要自己重新启动，那么以如下方式启动容器：

```
$> docker run <Run arguments> --restart=on-failure:5 <Container Image Name
or ID> <Command>
```

例如，

```
docker run --detach --restart=on-failure:5 nginx
```

影响

容器将尝试重新启动5次。

默认值

默认情况下，容器未配置重新启动策略。 因此，容器不会尝试重新启动它们自己。

参考文献

```
1. http://docs.docker.com/reference/commandline/cli/#restart-policies
```

5.15 不共享主机的进程名称空间（Scored）

配置适用性

- **Level 1- Docker**

描述

进程ID（PID）命名空间隔离进程ID号空间，这意味着不同PID名称空间中的进程可以具有相同的PID。 这是容器和主机之间的进程级隔离。

缘由

PID名称空间提供进程的分离。 PID命名空间删除了系统进程的视图，并允许包括PID 1在内的进程ID被重用。如果主机的PID名称空间与容器共享，基本上允许容器内的进程查看主机系统上的所有进程。 这打破了主机和容器之间进程级别隔离的好处。 有人访问容器可以最终知道主机系统上运行的所有进程，甚至可以从容器内杀死主机系统进程。 这可能是灾难性的。 因此，不要与容器共享主机的进程名称空间。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
PidMode={{.HostConfig.PidMode }}'
```

如果上述命令返回`“host”`，则表示主机PID名称空间与容器共享，否则符合规定。

修正

不要用`'--pid = host'`参数启动一个容器。

例如，不要以如下方式启动一个容器：

```
docker run --interactive --tty --pid=host centos /bin/bash
```

影响

容器进程无法看到主机系统上的进程。 在某些情况下，您希望您的容器共享主机的进程名称空间。例如，您可以使用`strace`或`gdb`等调试工具构建容器，但希望在调试容器内的进程时使用这些工具。 如果这是需要的，那么通过使用`“-p”`开关共享一个（或所需的）主机进程。

例如，

```
docker run --pid=host rhel7 strace -p 1234
```

默认值

默认情况下，所有容器都启用PID名称空间，并且主机的进程名称空间不与容器共享。

参考文献

```
1. https://docs.docker.com/reference/run/#pid-settings
2. http://man7.org/linux/man-pages/man7/pid_namespaces.7.html
```

5.16 不共享主机的IPC命名空间（Scored）

配置适用性

- **Level 1- Docker**

描述

IPC（POSIX / SysV IPC）命名空间提供了命名共享内存段，信号量和消息队列的分离。 因此，主机上的IPC命名空间不应与容器共享，并应保持独立。

缘由

IPC命名空间提供主机和容器之间的IPC分离。 如果主机的IPC名称空间与容器共享，基本上允许容器内的进程查看主机系统上的所有IPC。 这打破了主机和容器之间的IPC级隔离的好处。 有人访问容器可以最终操纵主机IPC。 这可能是灾难性的。 因此，不要与容器共享主机的IPC名称空间。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
IpcMode={{.HostConfig.IpcMode }}'
```

如果上述命令返回`“host”`，则表示主机IPC命名空间与容器共享。 如果上述命令不返回任何内容，则主机的IPC名称空间不共享。 这个符合规定。

修正

不要用`'--ipc = host'`参数启动一个容器。 例如，不要以如下方式启动一个容器：

```
docker run --interactive --tty --ipc=host centos /bin/bash
```

影响

共享内存段用于加速进程间通信。 它通常被高性能应用程序使用。 如果这样的应用程序被容器化为多个容器，则可能需要共享容器的IPC名称空间以实现高性能。 在这种情况下，您仍然应该共享容器特定的IPC名称空间，而不是主机IPC名称空间。 您可以与其他容器共享容器的IPC命名空间，如下所示：

```
$> docker run <Run arguments> --ipc=container:$INSTANCE_ID <Container Image
Name or ID> <Command>
```

例如，

```
docker run --interactive --tty --ipc=container:e3a7a1a97c58 centos
/bin/bash
```

默认值

默认情况下，所有容器都启用IPC命名空间，主机IPC命名空间不与任何容器共享。

参考文献

1. https://docs.docker.com/reference/run/#ipc-settings
2. http://man7.org/linux/man-pages/man7/namespaces.7.html

5.17 不要将主机设备直接暴露给容器（Not Scored）

配置适用性

- **Level 1- Docker**

描述

主机设备可以在运行时直接暴露给容器。 不要将主机设备直接暴露给容器，特别是对于不受信任的容器。

缘由

`--device`选项将主机设备暴露给容器，因此容器可以直接访问这些主机设备。 您不会要求容器以`privileged`模式运行以访问和操作主机设备。 默认情况下，容器将能够`read`，`write`和`mknod`这些设备。 此外，容器可能会从主机中删除块设备。 因此，不要直接将主机设备暴露给容器。 如果想把主机设备暴露给一个容器，适当地使用共享权限：

- r - read only
- w - writable
- m - mknod allowed

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
Devices={{.HostConfig.Devices }}'
```

以上命令将列出每个设备以下信息：

- `CgroupPermissions` - 例如, rwm
- `PathInContainer` - 容器内的设备路径
- `PathOnHost` - 主机上的设备路径

验证是否需要从容器中访问主机设备，并且正确设置所需的权限。 如果上述命令返回`[]`，那么容器不能访问主机设备。 这个可以被认为是合规的。

修正

不要将主机设备直接暴露于容器。 如果需要将主机设备暴露给容器，请使用正确的一组权限： 例如，不要以如下方式启动一个容器：

```
docker run --interactive --tty --device=/dev/tty0:/dev/tty0:rwm --
device=/dev/temp_sda:/dev/temp_sda:rwm centos bash
```

例如，以正确的权限共享主机设备：

```
docker run --interactive --tty --device=/dev/tty0:/dev/tty0:rw --
device=/dev/temp_sda:/dev/temp_sda:r centos bash
```

影响

您将无法直接在容器内使用主机设备。

默认值

默认情况下，没有主机设备暴露于容器。 如果您不提供共享权限并选择将主机设备公开到容器，则主机设备将具有`read`,`write`和`mknod`权限。

参考文献

```
1. http://docs.docker.com/reference/commandline/cli/#run
```

5.18 只在需要的时候在运行时覆盖默认的ulimit（Not Scored）

配置适用性

- **Level 1- Docker**

描述

默认的ulimit是在Docker daemon级别设置的。 但是，如果需要，您可以在容器运行时重写默认的ulimit设置。

缘由

`ulimit`提供了对shell可用的资源以及由它启动的进程的控制。 设置系统资源限制使您免于 fork炸弹等许多灾难。 有时候，即使友好的用户和合法的进程也会过度使用系统资源，从而导致系统无法使用。 应该遵守Docker daemon级别设置的默认ulimit。 如果默认的ulimit设置不适用于特定的容器实例，则可以将它们覆盖作为例外。 但是，不要这样做。 如果大部分容器实例都覆盖默认的ulimit设置，请考虑将默认的ulimit设置更改为适合您需要的设置。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
Ulimits={{.HostConfig.Ulimits }}'
```

上述命令应该为每个容器实例返回`Ulimits = <no value>`，除非有例外，并且需要覆盖默认的ulimit设置。

修正

如果需要，才覆盖默认的ulimit设置。 例如，要覆盖默认的ulimit设置，请按以下方式启动一个容器：

```
docker run --ulimit nofile=1024:1024 --interactive --tty centos /bin/bash
```

影响

如果ulimits设置不正确，可能无法实现所需的资源控制，甚至可能导致系统无法使用。

默认值

容器实例继承在Docker daemon级别设置的默认ulimit设置。

参考文献

```
1. http://docs.docker.com/reference/commandline/cli/#setting-ulimits-in-a-container
2. Command: man setrlimit
3. http://www.oreilly.com/webops-perf/free/files/docker-security.pdf
```

注意： ulimit中有多个可供选择的选项。 你可以控制的几个重要的限制是CPU（以秒为单位的CPU时间限制），fsize（文件进程可以创建的最大大小），memlock（可能锁定到RAM的最大内存字节数），nproc（mx进程数 ）等，完整列表可以见命令“man setrlimit”

5.19 不要将挂载的传播模式设为共享(Scored)

配置适用性

- **Level 1- Docker**

描述

挂载的传播允许在容器上以共享，从属或私有的模式来挂载卷，如果没有必要的话请不要使用共享模式来挂载卷。

缘由

共享挂载将在所有挂载中进行复制，并且将在任何挂载点进行的更改传播到所有安装。 以共享模式挂载卷不会限制其他容器的挂载并对该卷进行更改。 如果挂载的卷对变化敏感，这可能是灾难性的。 所以如果不是真的有必要的话不要将挂载的传播模式设为共享。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:Propagation={{range $mnt := .Mounts}} {{json $mnt.Propagation}} {{end}}'
```

上述命令将返回已挂载卷的传播模式。 除非需要，否则传播模式不应设置为“共享”。 如果没有挂载，上面的命令可能会引发错误。 在这种情况下，这个是不适用的。

修正

不要以共享的传播模式挂载卷，例如，不要以如下方式启动容器：

```
docker run <Run arguments> --volume=/hostPath:/containerPath:shared <Container Image Name or ID> <Command>
```

影响

无

默认值

默认情况下，容器的挂载都是私有的。

参考文献

```
1. https://github.com/docker/docker/pull/17034
2. https://docs.docker.com/engine/reference/run/
3. https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt
```

5.20 不要共享主机的UTS命名空间（Scored）

配置适用性

- **Level 1- Docker**

描述

UTS命名空间提供两个系统标识符的隔离：主机名和NIS域名。 它用于设置在该名称空间中正在运行的进程可见的主机名和域。 在容器中运行的进程通常不需要知道主机名和域名。 因此，命名空间不应与主机共享。

缘由

与主机共享UTS命名空间会对容器提供完全权限以更改主机的主机名。 这是不安全的，不应该被允许。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
UTSMode={{.HostConfig.UTSMode }}'
```

如果上述命令返回`“host”`，则意味着主机UTS名称空间与容器共享，并且不符合基线。 如果上述命令不返回任何内容，则主机的UTS名称空间不会共享。 是符合这个基线的。

修正

不要使用`“--uts = host”`参数启动容器。 例如，不要以如下方式启动一个容器：

```
docker run --rm --interactive --tty --uts=host rhel7.2
```

影响

无

默认值

默认情况下，所有容器都启用了UTS命名空间，并且主机UTS命名空间不与任何容器共享。

参考文献

```
1. https://docs.docker.com/engine/reference/run/
2. http://man7.org/linux/man-pages/man7/namespaces.7.html
```

5.21 不要禁用默认seccomp配置文件（Scored）

配置适用性

- **Level 1- Docker**

描述

Seccomp过滤为进程指定传入系统调用的过滤器提供了一种手段。 默认的Docker seccomp配置文件基于白名单，并允许311个系统调用阻止所有其他的配置。 它不应该被禁用，除非它阻碍你的容器应用程序的使用。

缘由

大量的系统调用暴露于每个用户级进程，其中许多系统调用在整个生命周期中都未被使用。 大多数应用程序不需要所有的系统调用，因此可以通过减少可用的系统调用来获益。 减少的一组系统调用减少了暴露给应用程序的整个内核表面，从而提高应用程序的安全性。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
SecurityOpt={{.HostConfig.SecurityOpt }}'
```

上面的命令应该返回`“<no value>”`或您修改的seccomp配置。 如果它返回`[seccomp: unconfined]`，这意味着不符合此基线，容器没有使用任何seccomp配置文件。

修正

默认情况下，seccomp配置文件已启用。 除非要修改和使用修改后的seccomp配置文件，否则不需要执行任何操作。

影响

使用Docker 1.10或更高版本，默认的seccomp配置文件阻塞系统调用，而不管传递给容器的`--cap-add`。 在这种情况下，您应该创建自己的自定义seccomp配置文件。 您也可以通过在`docker run`上传递`--security- opt = seccomp: unconfined`来禁用默认的seccomp配置文件。

默认值

运行容器时，除非使用`--security-opt`选项覆盖，否则它将使用默认配置文件。

参考文献

```
1. http://blog.scalock.com/new-docker-security-features-and-what-they-mean-seccomp-profiles
2. https://docs.docker.com/engine/reference/run/
3. https://github.com/docker/docker/blob/master/profiles/seccomp/default.json
4. https://docs.docker.com/engine/security/seccomp/
5. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
6. https://github.com/docker/docker/issues/22870
```

5.22 不要执行具有特权选项的docker exec命令（Scored）

配置适用性

- **Level 2- Docker**

描述

不要使用`--privileged`选项执行`docker exec`。

缘由

在`docker exec`中使用`--privileged`选项可为命令提供扩展的Linux功能。 这可能是不安全的，特别是在运行具有丢弃功能或增强限制的容器时。

审计

如果按照第1节的规定启用了审计，则可以使用以下命令过滤掉使用`--privileged`选项的`docker exec`命令。

```
ausearch -k docker | grep exec | grep privileged
```

修正

在`docker exec`命令中不要使用`--privileged`选项。

影响

无。 如果您需要在容器内增强功能，请使用所需功能运行容器。

默认值

默认情况下，docker exec命令运行时没有`--privileged`选项。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/exec/
```

5.23 不要使用user选项的执行docker exec命令（Scored）

配置适用性

- **Level 2- Docker**

描述

docker exec不要使用`--user`选项。

缘由

在`docker exec`中使用`--user`选项以该用户身份执行容器内的命令。 这可能是不安全的，特别是在运行具有丢弃功能或增强限制的容器时。 例如，假设你的容器是以tomcat用户（或任何其他非root用户）的身份运行的，那么可以通过`docker exec` `--user = root`选项以`root`身份运行一个命令。 这可能是危险的。

审计

如果您按照第1节的规定启用了审计，则可以使用以下命令过滤掉使用了`--user`选项的`docker exec`命令。

```
ausearch -k docker | grep exec | grep user
```

修正

不要使用`--user`选项执行`docker exec`

影响

无

默认值

默认情况下，执行`docker exec`时，没有使用`--user`选项。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/exec/
```

5.24 确认cgroup的用法(Scored)

配置适用性

- **Level 1- Docker**

描述

可以在容器运行时附加到特定的cgroup，确认cgroup使用将确保容器在定义的cgroup下运行。

缘由

系统管理员通常定义容器应该在哪个cgroup下运行。 即使cgroups没有被系统管理员明确定义，默认情况下，容器会在docker cgroup下运行。 在运行时，可以附加到不同于预期使用的cgroup之外的其他cgroup。 这个用法应该被监视和确认。 通过附加到一个不同的cgroup，可能会被授予额外的权限和资源，因此可能被证明是不安全的。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
CgroupParent={{.HostConfig.CgroupParent }}'
```

上述命令将返回容器正在运行的cgroup。 如果它是空的，这意味着容器在默认的`docker cgroup`下运行。 在这种情况下，是符合基线的。 如果发现容器正在运行非预期的cgroup，是不符合此基线的。

修正

除非需要，否则不要在docker run命令中使用`--cgroup-parent`选项。

影响

无

默认值

默认情况下，容器在`docker` cgroup下运行。

参考文献

```
1. https://docs.docker.com/engine/reference/run/#specifying-custom-cgroups
2.https://access.redhat.com/documentation/en_US/Red_Hat_Enterprise_Linux/6/h
tml/Resource_Management_Guide/ch01.html
```

5.25 限制容器获得额外的权限（得分）

配置适用性

- **Level 1- Docker**

描述

限制容器通过`suid`或`sgid`位获得额外的权限。

缘由

一个进程可以在内核中设置`no_new_priv`位。其值贯穿于`fork`，`clone`和`execve`。`no_new_priv`位确保进程或其子进程不会通过`suid`或`sgid`位获得任何额外的权限。这样，很多危险的操作变得不那么危险，因为不可能破坏特权二进制文件。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
SecurityOpt={{.HostConfig.SecurityOpt }}'
```

上述命令应该返回当前为容器配置的所有安全选项。 `no-new-privileges`也应该是其中之一。

修正

以如下方式启动容器：

```
docker run <run-options> --security-opt=no-new-privileges <IMAGE> <CMD>
```

例如，

```
docker run --rm -it --security-opt=no-new-privileges ubuntu bash
```

影响

`no_new_priv`防止像SELinux这样的LSM转换到不允许访问当前进程的进程标签。

默认值

默认情况下，新的权限不受限制。

参考文献

```
1. https://github.com/projectatomic/atomic-site/issues/269
2. https://github.com/docker/docker/pull/20727
3. https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt
4. https://lwn.net/Articles/475678/
5. https://lwn.net/Articles/475362/
```

5.26 在容器运行时对容器做健康检查(Scored)

配置适用性

- **Level 1- Docker**

描述

如果容器镜像没有定义`HEALTHCHECK`指令，则在容器运行时使用`--health-cmd`参数来检查容器运行状况。

缘由

安全的三要素之一是可用性。如果您正在使用的容器镜像没有预定义的`HEALTHCHECK`指令，请使用`--health-cmd`参数在运行时检查容器运行状况。 根据报告的健康状况，您可以采取必要的措施。

审计

运行以下命令并确保所有容器都报告运行状况：

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: Health=
{{.State.Health.Status }}'
```

修正

使用`--health-cmd`和其他参数运行容器。 例如，

```
docker run -d --health-cmd='stat /etc/passwd || exit 1' nginx
```

影响

无

默认值

默认情况下，容器运行时并没有做健康检查。

参考文献

```
1. https://github.com/docker/docker/pull/22719
```

5.27 确保docker命令始终获取最新版本的镜像（Not Scored）

配置适用性

- **Level 1- Docker**

描述

始终确保在存储库中使用最新版本的映像，而不是缓存的旧版本。

缘由

多个docker命令，比如`docker pull`，`docker run`等，存在一个已知问题，即默认情况下它们会提取镜像的本地副本（如果存在的话），即使在上游存储库中存在具有“相同标记”的更新版本的镜像“。 这可能会导致使用较老的和易受攻击的镜像。

审计

步骤1：打开您的镜像库，并列出您正在检查的镜像的镜像版本历史记录。

步骤2：观察`docker pull`命令启动时的状态，如果状态显示为`Image is up to date,`，则表示您正在获取的镜像是缓存版本。

步骤3：将正在运行的镜像版本与存储库中报告的最新版本匹配，以确定您是在运行缓存版本还是最新版本。

修正

使用适当的版本固定机制（默认分配的最新标记仍然容易受到缓存攻击），以避免提取缓存的旧版本。 版本固定机制也应该用于基本映像，软件包和整个映像。 您可以根据您的要求自定义版本固定规则。

影响

无

默认值

默认情况下，除非使用了版本固定机制或清除了本地缓存，否则docker命令会提取本地副本。

参考文献

```
1. https://github.com/docker/docker/pull/16609
```

5.28 使用PID cgroup限制（Scored）

配置适用性

- **Level 1- Docker**

描述

在容器运行时使用`--pids-limit`标志

缘由

攻击者可以在容器内用单个命令形成fork炸弹。 这种fork炸弹可能会使整个系统崩溃，并需要重新启动主机才能使系统再次正常运行。 PID cgroup `--pids-limit`将通过限制在给定时间内容器内可能发生的fork数量来防止这种攻击。

审计

运行以下命令并确保`PidsLimit`未设置为`0`或`-1`。`0`或`-1`的`PidsLimit`表示可以同时在容器内fork任意数量的进程。

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
PidsLimit={{.HostConfig.PidsLimit }}'
```

修正

使用`--pids-limit`标志，以适当值启动容器。 例如，

```
 docker run -it --pids-limit 100 <Image_ID>
```

在上面的例子中，在任何给定的时间允许运行的进程数被设置为100.在达到100个并发运行进程的限制之后，docker将限制任何新的进程创建。

影响

请根据需要设置PID值。 否则可能会使容器无法使用。

默认值

`--pids-limit`的默认值是0，这意味着fork的数量没有限制。 另外请注意，PID cgroup限制仅适用于内核版本4.3及以上的版本

参考文献

```
1. https://github.com/docker/docker/pull/18697
2. https://docs.docker.com/engine/reference/commandline/run/
```

5.29 不要使用Docker的默认网桥docker0（Not Scored）

配置适用性

- **Level 2- Docker**

描述

不要使用Docker的默认网桥`docker0`。 使用用户定义的docker网络进行容器联网

缘由

Docker将以桥模式创建的虚拟接口连接到一个名为`docker0`的公共桥。 由于没有使用过滤，这种默认的网络模型容易受到ARP欺骗和MAC洪泛攻击。

审计

运行以下命令，并验证容器是否在用户定义的网络上，而不是默认的`docker0`网桥。

```
docker network ls --quiet | xargs xargs docker network inspect --format '{{
.Name }}:{{ .Options }}'
```

修正

遵循Docker文档并设置用户定义的网络。 在定义的网络中运行所有容器。

影响

您必须管理用户自定义的网络。

默认值

默认情况下，docker在其`docker0`桥上运行容器。

参考文献

```
1. https://github.com/nyantec/narwhal
2. https://arxiv.org/pdf/1501.02967
3. https://docs.docker.com/engine/userguide/networking/dockernetworks/
```

5.30 不要共享主机的用户名空间（Socred）

配置适用性

- **Level 1- Docker**

描述

不要与容器共享主机的用户名称空间。

缘由

用户名空间确保容器内的root进程将映射到容器外的非root进程。 因此，使用容器共享主机的用户命名空间不会将容器上的用户与主机上的用户隔离开来。

审计

运行下面的命令不会返回`UsernsMode`的任何值。 如果它返回一个`host`的值，这意味着主机用户名称空间与容器共享。

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
UsernsMode={{ .HostConfig.UsernsMode }}'
```

修正

不要在主机和容器之间共享用户名空间。

影响

无

默认值

默认情况下，主机用户名称空间与容器没有启用共享。

参考文献

```
1. https://docs.docker.com/engine/reference/commandline/run/#/run
2.
https://events.linuxfoundation.org/sites/events/files/slides/User%20Namespac
es%20-%20ContainerCon%202015%20-%2016-9-final_0.pdf
3. https://github.com/docker/docker/pull/12648
```

5.31 不要将Docker套接字挂载到任何容器中（Scored）

配置适用性

- **Level 1- Docker**

描述

docker套接字（`docker.sock`）不应该挂载到容器内。

缘由

如果docker套接字被挂载到一个容器内，它将允许在容器中运行的进程执行可以完全控制docker主机的命令。

审计

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
Volumes={{ .Mounts}}' | grep docker.sock
```

以上命令将返回`docker.sock`已经映射到容器之内的所有docker实例。

修正

确保没有容器将docker.sock挂载为卷。

影响

无

默认值

默认情况下，`docker.sock`不会挂载到容器中。

参考文献

```
1. https://raesene.github.io/blog/2016/03/06/The-Dangers-Of-Docker.sock/
2. https://forums.docker.com/t/docker-in-docker-vs-mounting-var-run-docker-
sock/9450/2
3. https://github.com/docker/docker/issues/21109
```

6 安全操作

本章节介绍了Docker部署的一些操作安全方面的东西。这些是应该遵循的最佳实践。 这里的大部分建议是只是提醒组织应该延长当前的安全最佳实践和政策包括容器。

6.1 定期对您的主机系统和容器进行安全审计（Not Scored）

配置适用性

- **Level 1- Docker**

描述

定期对您的主机系统和容器进行安全审计，以确定可能使您的系统受到危害的任何错误配置或漏洞。

缘由

对主机系统和容器执行定期和专门的安全审计，可以提供那些在日常工作中你可能不了解的深入性的安全见解，安全漏洞的鉴定会比较轻松，而且这一切都会提高你的环境的安全性。

审计

遵循你所在组织的安全审计策略和要求。

修正

遵循你所在组织的安全审计策略和要求。

影响

无

默认值

不适用

参考文献

```
1. http://searchsecurity.techtarget.com/IT-security-auditing-Best-practices-for-conducting-audits
```

6.2 监测Docker容器的使用，性能和计量（metering）（Not Scored）

配置适用性

- **Level 1- Docker**

描述

容器可能运行对您的业务至关重要的服务。 监测他们的使用，性能和计量将是非常重要的。

缘由

跟踪容器的使用、性能和和相关的计量数据，在你使用容器来运行关键服务时，这对你来说很关键，它会给带来：

- 容量管理和优化
- 性能管理
- 全面可视化

容器性能的这种深度可见性将帮助您确保容器的高可用性和最小的停机时间。

审计

```
docker stats $(docker ps --quiet)
```

上述命令将为每个正在运行的容器返回CPU，内存和网络统计信息。

修正

使用软件或容器来跟踪容器使用情况，报告性能和计量。

影响

要获得容器指标，您将不得不在特权模式下使用另一个容器，或者使用可进入各种容器名称空间的软件。 对所有容器的名称空间提供不受限制的访问可能太冒险了。

默认值

默认情况下，对于每个容器，系统通过执行控制组（cgroups）来跟踪有关CPU，内存和块I / O使用情况的运行时度量标准，如下所示：

```
CPU - /sys/fs/cgroup/cpu/system.slice/docker-$INSTANCE_ID.scope/
Memory - /sys/fs/cgroup/memory/system.slice/docker-$INSTANCE_ID.scope/
Block I/O - /sys/fs/cgroup/blkio/system.slice/docker-$INSTANCE_ID.scope/
```

参考文献

```
1. https://docs.docker.com/articles/runmetrics/
2. https://github.com/google/cadvisor
3. https://docs.docker.com/reference/commandline/cli/#stats
```

6.3 备份容器数据（Not Scored）

配置适用性

- **Level 1- Docker**

描述

定期对容器的数据卷进行备份

缘由

容器可能运行着对你的业务至关重要的服务，定期备份数据会保证如果有任何数据的丢失的话你仍然有备份数据。而数据的真的丢失的话，很有可能毁掉你的业务。

审计

询问系统管理员是否定期备份容器数据卷。 验证备份的副本并确保遵循组织的备份策略。 另外，您可以对每个容器实例执行以下命令，以在容器文件系统中列出已更改的文件和目录。 理想情况下，容器的文件系统上不应该存储任何内容。

```
docker diff $INSTANCE_ID
```

修正

您应该遵循组织的数据备份策略。 您可以使用`'--volumes-from'`参数备份您的容器数据卷，如下所示：

```
$> docker run <Run arguments> --volumes-from $INSTANCE_ID -v [host-dir]:
[container- dir] <Container Image Name or ID> <Command>
```

例如，

```
docker run --volumes-from 699ee3233b96 -v /mybackup:/backup centos tar cvf
/backup/backup.tar /exampledatatobackup
```

影响

无

默认值

默认情况下，容器数据卷不会进行数据备份。

参考文献

```
1. http://docs.docker.com/userguide/dockervolumes/#backup-restore-or-
migrate-data-volumes
2. http://stackoverflow.com/questions/26331651/back-up-docker-container-
that-has-a-volume
3. http://docs.docker.com/reference/commandline/cli/#diff
```

6.4避免镜像蔓延（Not Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

不要在同一个主机上保留大量的容器镜像。 根据需要只使用标记的镜像。

缘由

标记的镜像有助于从“最新”回滚到生产中特定版本。 未使用或旧标记的镜像可能包含可能被利用的漏洞（如果实例化）。 此外，如果您无法从系统中删除未使用的映像，并且存在各种冗余和未使用的映像，主机文件系统可能会变满，并可能导致拒绝服务。

审计

步骤1：通过执行下面命令列出所有当前实例化的镜像ID：

```
docker images --quiet | xargs docker inspect --format '{{ .Id }}: Image=
{{.Config.Image }}'
```

步骤2：通过执行以下命令列出系统中存在的所有镜像：

```
docker images
```

步骤3：比较步骤1和步骤2中的镜像ID列表，找出当前未被实例化的镜像。 如果发现任何此类未使用或旧镜像，请与系统管理员讨论是否需要将这些镜像保存在系统中。 如果不满足，不符合此基线。

修正

保留您实际需要的一组镜像，并建立一个工作流，以从主机中删除旧的或陈旧的镜像。 此外，使用如摘要的功能从仓库中获取特定的镜像。 另外，您可以按照下面的步骤来找出系统中未使用的镜像并将其删除。

步骤1：执行如下命令列出所有的近期实例化的镜像的ID：

```
docker images --quiet | xargs docker inspect --format '{{ .Id }}: Image=
{{.Config.Image }}'
```

步骤2：执行如下命令列出系统中当前所有的镜像:

```
docker images
```

步骤3：对比步骤1和步骤2中列出的镜像并且找出近期未实例化过得镜像。

步骤4：决定是否继续保留最近未使用的镜像，如果你想删除它们的话执行如下命令：

```
docker rmi $IMAGE_ID
```

影响

无

默认值

在主机上，在管理员移除所有的与镜像和分层的文件系统相关的标签前，它们仍然是可用的。

参考文献

1. http://craiccomputing.blogspot.in/2014/09/clean-up-unused-docker-containers-and.html
2. https://forums.docker.com/t/command-to-remove-all-unused-images/20/8
3. https://github.com/docker/docker/issues/9054
4. http://docs.docker.com/reference/commandline/cli/#rmi
5. http://docs.docker.com/reference/commandline/cli/#pull
6. https://github.com/docker/docker/pull/11109

6.5 避免容器蔓延（Not Scored）

配置适用性

- **Level 1- Linux Host OS**

描述

不要在同一个主机上保留大量的容器。

缘由

容器的灵活性使得运行多个应用程序的实例变得很容易，并间接导致存在不同安全补丁级别的Docker镜像。 这也意味着你正在消耗主机资源。 在特定的主机上有不止一个容器的可管理数量，这使得这种情况容易受到错误处理，配置错误和碎片的影响。 因此，避免容器蔓延，并将主机上的容器数量控制在一个可管理的总量上。

审计

执行`docker info`得到主机上的容器数量：

```
docker info --format '{{.Containers}}'
```

现在，执行如下命令得到主机上的正在运行的或者已经停止的容器。

```
docker info --format '{{.ContainersRunning}}'
docker info --format '{{.ContainersStopped}}'
```

如果主机上已经停止的容器数量大于正在运行的容器的数量太多（25或者更多）的话，该主机上的容器很有可能已经被蔓延了。

修正

定期检查每个主机的容器清单，并使用以下命令清理已停止的容器：

```
docker container prune
```

影响

如果每个主机的容器数量太少，那么也许你没有充分利用你的主机资源。

默认值

默认情况下，Docker不会限制主机上的容器数量。

参考文献


```
1. https://zeltser.com/security-risks-and-benefits-of-docker-application/
2. http://searchsdn.techtarget.com/feature/Docker-networking-How-Linux-containers-will-change-your-network
```