Docker逃逸与防护策略

主讲人：张谦

目录

Docker简介

Docker简介

什么是 Docker

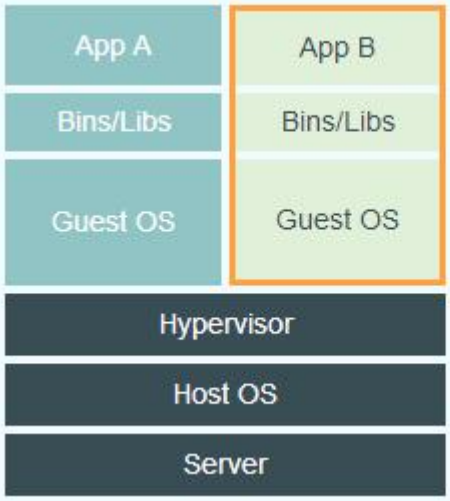
Docker 是一个开源项目，诞生于 2013 年初，基于 Google 公司推出的 Go 语言实现。

Docker 项目的目标是实现轻量级的操作系统虚拟化解决方案。

基础是 Linux 容器（LXC）等技术。在 LXC 的基础上 Docker 进行了进一步的封装，用户操作 Docker 的容器就像操作一个快速轻量级的虚拟机一样简单。

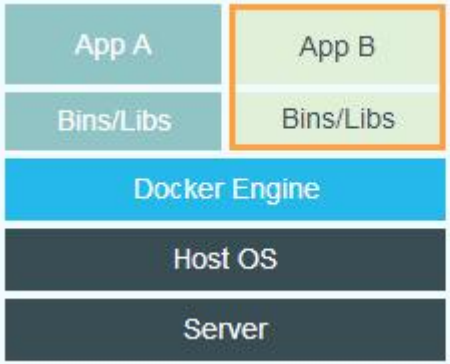
下面的图片比较了 Docker 和传统虚拟化方式的不同之处：

容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，而传统方式则是在硬件层面实现。



Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

Docker简介-应用场景

一台 16 核 32G 内存的主机，需要跑 500+ 个用户的应用，每个应用的功能可以认为是一个网站 ，有两个事情很重要：

1.资源隔离：比如限制应用最大内存使用量，或者资源加载隔离等。

2.低消耗：虚拟化本身带来的损耗需要尽量的低。

我们不可能在一台机器上开 500 个虚拟机，虽然可以在资源隔离方面做的很好，但这种虚拟化本身带来的资源消耗太严重。

而 Docker 很好的权衡了两者，即拥有不错的资源隔离能力，又有很低的虚拟化开销。

Docker核心技术

——Kernel Namespace

——Control Groups

——Another Union FS (AUFS)

Docker核心技术-Kernel namespace

Namespaces提供了最直接有效的资源隔离：
进程在不同容器以及宿主机中互不可见

Linux内核提供了下面6种NameSpace:

Namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

Docker核心技术-Kernel namespace

```
//内核中描述每个进程的数据结构
struct task_struct {
        . . .
        /* open file information */
        struct files_struct *files;

        /* namespaces */
        struct nsproxy *nsproxys;
        . . .
};

//该结构中有一个nsproxy成员包含了5个namespaces的指针：
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
};
```

Docker核心技术-Kernel namespace

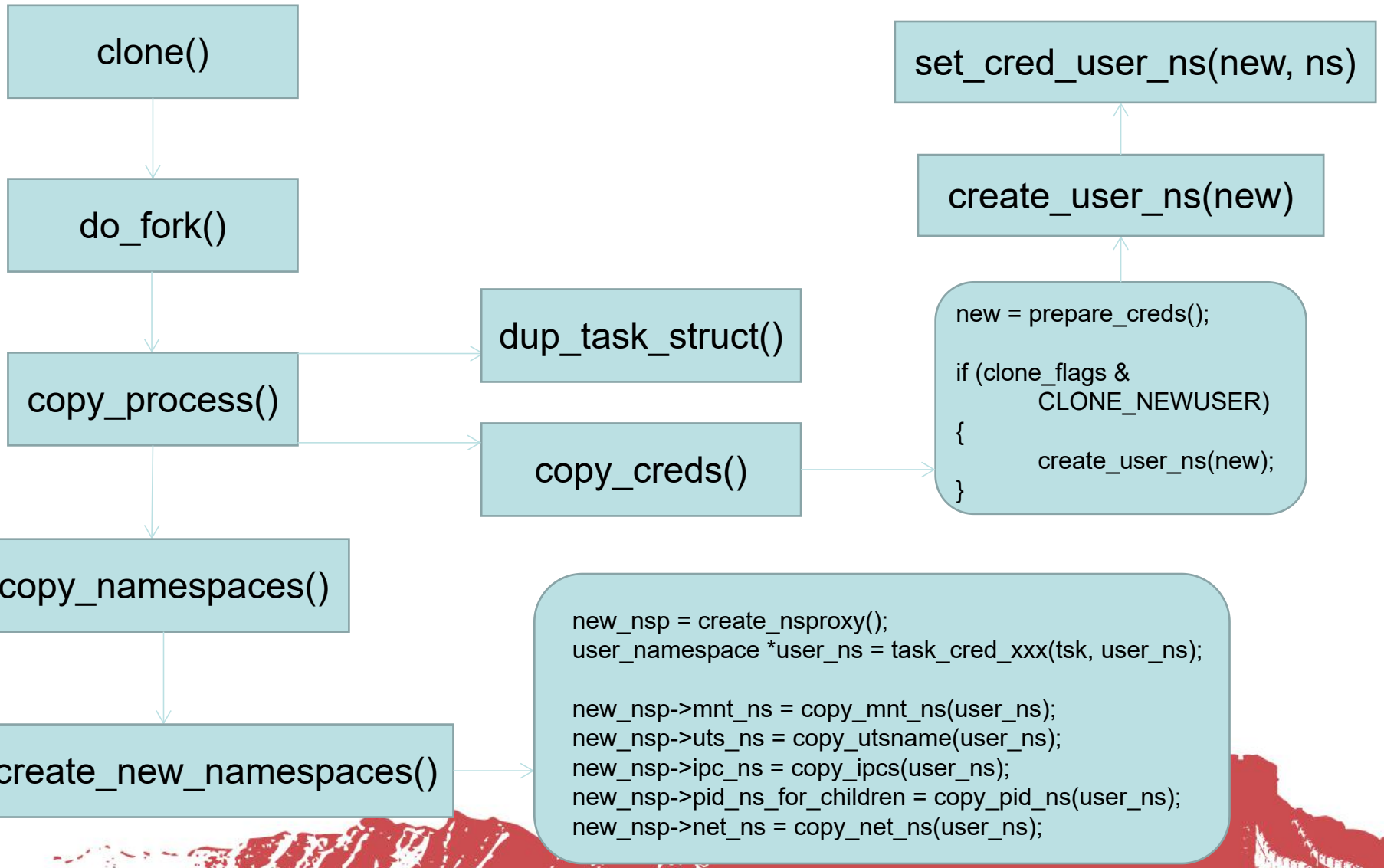
在Linux下，用来创建一个新的进程需要用到clone()，原型如下：

long clone(unsigned long *flags*, void **child_stack*, void **ptid*, void **ctid*, struct pt_regs **regs*);

其中参数flags可以用表格里的参数进行按位或，以表明子进程需要创建哪些Namespace。

接下来从源码来分析clone系统调用是怎样把各种Namespace创建出来的。

Docker核心技术-Kernel namespace



Docker核心技术-Kernel namespace

```
static void set_cred_user_ns(struct cred *cred, struct user_namespace
*user_ns)
{
        cred->securebits = SECUREBITS_DEFAULT;
        cred->cap_inheritable = CAP_EMPTY_SET;
        cred->cap_permitted = CAP_FULL_SET;
        cred->cap_effective = CAP_FULL_SET;
        cred->cap_bset = CAP_FULL_SET;

#ifdef CONFIG_KEYS
        key_put(cred->request_key_auth);
        cred->request_key_auth = NULL;
#endif

        cred->user_ns = user_ns;
}
```

Docker核心技术-资源隔离的实现

以UTS Namespace举个简单例子，来看看内核是如何实现资源隔离的。

```
struct uts_namespace {
        struct kref kref;
        struct new_utsname name;
        struct user_namespace *user_ns;
        unsigned int proc_inum;
};

struct new_utsname {
        char sysname[__NEW_UTS_LEN + 1];
        char nodename[__NEW_UTS_LEN + 1];
        char release[__NEW_UTS_LEN + 1];
        char version[__NEW_UTS_LEN + 1];
        char machine[__NEW_UTS_LEN + 1];
        char domainname[__NEW_UTS_LEN + 1];
};
```

Docker核心技术-资源隔离的实现

```
SYSCALL_DEFINE2(sethostname, char __user *, name, int, len)
{
        int errno;
        char tmp[__NEW_UTS_LEN];

        //检查当前进程是否具有权限修改hostname
        if (!ns_capable(current->nsproxy->uts_ns->user_ns, CAP_SYS_ADMIN))
                return -EPERM;

        if (len < 0 || len > __NEW_UTS_LEN)
                return -EINVAL;

        //将用户态hostname拷贝到内核
        if (!copy_from_user(tmp, name, len)) {
                struct new_utsname *u = &current->nsproxy->uts_ns->name;

                memcpy(u->nodename, tmp, len);
                memset(u->nodename + len, 0, sizeof(u->nodename) - len);
                errno = 0;
                uts_proc_notify(UTS_PROC_HOSTNAME);
        }

        return errno;
}
```

Docker核心技术-资源隔离的实现


```
SYSCALL_DEFINE2(gethostname, char __user *, name, int, len)
{
        int i, errno;
        struct new_utsname *u;

        if (len < 0)
                return -EINVAL;

        u = &current->nsproxy->uts_ns->name;
        i = 1 + strlen(u->nodename);
        if (i > len)
                i = len;
        errno = 0;

        //拷贝hostname到用户空间
        if (copy_to_user(name, u->nodename, i))
                errno = -EFAULT;

        return errno;
}
```

Docker核心技术-权限检查

以刚才sethostname为例，那么则要求当前进程在所请求的user_namespace中具有CAP_SYS_ADMIN权限：

```
if (!ns_capable(current->nsproxy->uts_ns->user_ns, CAP_SYS_ADMIN))
                        return -EPERM;


bool ns_capable(struct user_namespace *ns, int cap)
{
        if (unlikely(!cap_valid(cap))) {
                pr_crit("capable() called with invalid cap=%u\n", cap);
                BUG();
        }

        if (security_capable(current_cred(), ns, cap) == 0) {
                current->flags |= PF_SUPERPRIV;
                return true;
        }
        return false;
}
```

Docker核心技术-权限检查

```
int cap_capable(const struct cred *cred, struct user_namespace *targ_ns,
                int cap, int audit)
{
        struct user_namespace *ns = targ_ns;

        for (;;) {
                if (ns == cred->user_ns)
                        return cap_raised(cred->cap_effective, cap) ? 0 : -EPERM;

                if (ns == &init_user_ns)
                        return -EPERM;

                //检查当前进程的user_ns是否是目的user_ns的父namespace
                if ((ns->parent == cred->user_ns) && uid_eq(ns->owner, cred->euid))
                        return 0;

                ns = ns->parent;
        }

        /* We never get here */
}
```

Docker核心技术-NS相关系统调用

创建Namespace的系统调用包括clone(), unshare(), setns()

——int unshare(int flags);

调用unshare()的主要作用就是不启动一个新进程就可以起到隔离的效果，
相当于跳出原先的namespace进行操作。这样，你就可以在原进程进行一些需要隔离的操作。

——int setns(int fd, int nstype);

通过setns()加入一个已经存在的namespace
参数fd表示我们要加入的Namespace的文件描述符。
它是一个指向/proc/[pid]/ns目录的文件描述符，
可以通过直接打开该目录下的链接或者打开一个挂载了该目录下链接的文件得到。

Docker核心技术-NS相关系统调用

从3.8版本的内核开始，用户就可以在/proc/[pid]/ns文件下看到指向不同namespace号的文件，效果如下所示，形如[4026531835]即为namespace号。

```
vsec@vsec-virtual-machine:~$ ll /proc/$$/ns
total 0
dr-x--x--x 2 vsec vsec 0 8月  12 13:44 ./
dr-xr-xr-x 9 vsec vsec 0 8月  12 13:44 ../
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 net -> net:[4026531957]
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 pid -> pid:[4026531836]
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 user -> user:[4026531837]
lrwxrwxrwx 1 vsec vsec 0 8月  12 13:44 uts -> uts:[4026531838]
```

如果两个进程指向的namespace编号相同，就说明他们在同一个namespace下，否则则在不同namespace里面

Docker核心技术

——Kernel Namespace

——Control Groups

——Another Union FS (AUFS)

Docker核心技术-Control Groups

CGroups是 Linux 内核的一个特性，主要用来对共享资源进行隔离、限制、审计等。

只有控制分配到容器的资源，才能避免当多个容器同时运行时的对系统资源的竞争，确保了当容器内的资源使用产生压力时不会连累主机系统

控制组技术最早是由 Google 的程序员 2006 年起提出，Linux 内核自 2.6.24 开始支持。

控制组可以提供对容器的内存、CPU、磁盘 IO 等资源的限制和审计管理。

CGroups 的使用非常简单，提供类似文件的接口，在 /sys/fs/cgroup目录下新建一个文件夹即可新建一个 cgroup，在此文件夹中新建task文件，并将pid写入该文件，即可实现对该进程的资源控制

Docker核心技术-Control Groups

CGroups可以限制10大子系统的资源，以下是每个子系统的详细说明：

1. blkio 这个子系统设置限制每个块设备的输入输出控制。例如:磁盘，光盘以及usb等等。

2. cpu 这个子系统使用调度程序为cgroup任务提供cpu的访问。

3. cpuacct 产生cgroup任务的cpu资源报告。

4. cpuset 如果是多核心的cpu，这个子系统会为cgroup任务分配单独的cpu和内存。

5. devices 允许或拒绝cgroup任务对设备的访问。

6. freezer 暂停和恢复cgroup任务。

7. memory 设置每个cgroup的内存限制以及产生内存资源报告。

8. net_cls 标记每个网络包，允许linux traffic controller 识别从特定cgroup发出的包。

9. net_prio 动态设置每个网络接口的优先级。

10. ns 名称空间子系统。

Docker核心技术-Control Groups

术语解释：

cgroup – 将一组task和一组subsystem的配置参数关联，cgroup是资源分片的最小单位。

subsystem – 子系统，其实就是资源管理器，上面介绍过。

hierarchy – 层级，可以理解为cgroup及subsystem集合的根目录。有些发行版默认将每个subsystem独自挂载为一个层级.也可以将多个subsystem挂载为一个层级：

```
mount -t cgroup -o cpu,cpuset,memory cpu_and_mem
/sys/fs/cgroup/cpu_and_mem
```

Docker核心技术-Control Groups

一个典型的hierarchy目录如下：

```
/sys/fs/cgroup/
├── cpu                                  <------ hierarchy/root cgroup
│   ├── cgroup.clone_children            <------ subsystem 参数
│   ├── cgroup.procs
│   ├── cpuacct.stat
│   ├── cpuacct.usage
│   ├── cpuacct.usage_percpu
│   ├── vsec                             <------ cgroup
│   │   ├── cpu.cfs_quota_us             <------ subsystem 参数
│   │   ├── cpu.cfs_period_us
... ... ...
│   │   └── tasks                        <-------task list
│   ├── notify_on_release
│   ├── release_agent
│   └── tasks
...
```

Docker核心技术-Control Groups

#cgroups管理进程cpu资源

有如下脚本：

```
x=0
while [ True ];do
        x=$x+1
done;
```

用top可以看到这个脚本占用了100%的CPU

```
 PID   USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+  COMMAND
 6814 tyrande+  20   0    6936   3364   1432 R  100.0  0.1        0:10.82  cpu.sh
```

下面用cgroup控制该进程所占用CPU资源

```
root@ubuntu:/sys/fs/cgroup/cpu/cg1# echo 50000 > cpu.cfs_quota_us
root@ubuntu:/sys/fs/cgroup/cpu/cg1# echo 6814 > tasks
```

再次用top查看实时数据，发现CPU占用率降为50%

```
 PID   USER      PR  NI    VIRT    RES    SHR  S  %CPU %MEM     TIME+   COMMAND
 6814 tyrande+  20   0   12888   8008   1432 R  49.8   0.2        2:07.11 cpu.sh
```

Docker核心技术

——Kernel Namespace

——Control Groups

——Another Union FS (AUFS)

Docker核心技术-AUFS

联合文件系统（UnionFS）是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加

联合文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

root@vsec-virtual-machine:/home/vsec# docker commit -m "test" -a "docker newbee" 62b9554a9fae test/test:v2
sha256:9e77ccb63f56d5397790ac220736d108603cfbec7ed83075985b1c5f01445a80

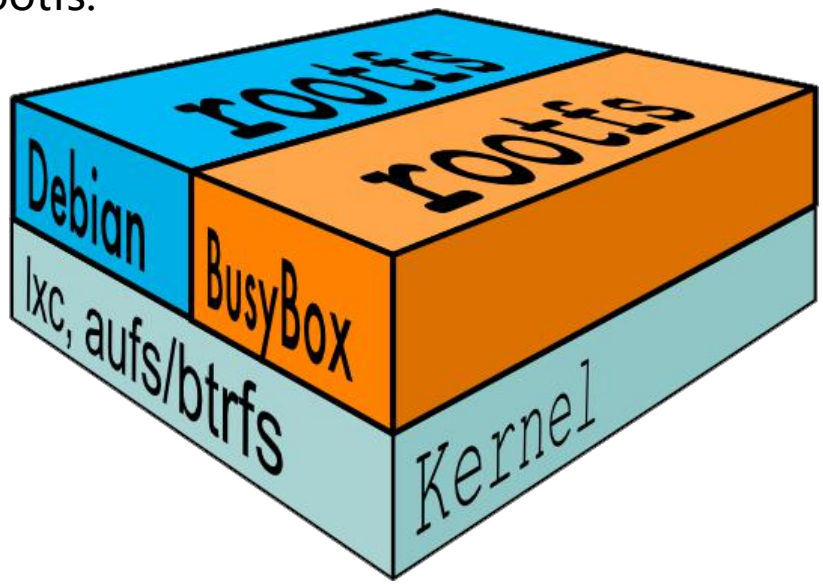
Docker 中使用的 AUFS（Another Union FS）就是一种联合文件系统。 AUFS 支持为每一个成员目录设定只读（readonly）、读写（readwrite）和写出（whiteout-able）权限, 同时 AUFS 里有一个类似分层的概念, 对只读权限的分支可以逻辑上进行增量地修改(不影响只读部分的)。

Docker核心技术-AUFS

bootfs (boot file system)
主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel, 当boot成功后 kernel 被加载到内存中后 bootfs就被umount了.

rootfs (root file system)
包含的就是典型 Linux 系统中的 /dev, /proc, /bin, /etc 等标准目录和文件。

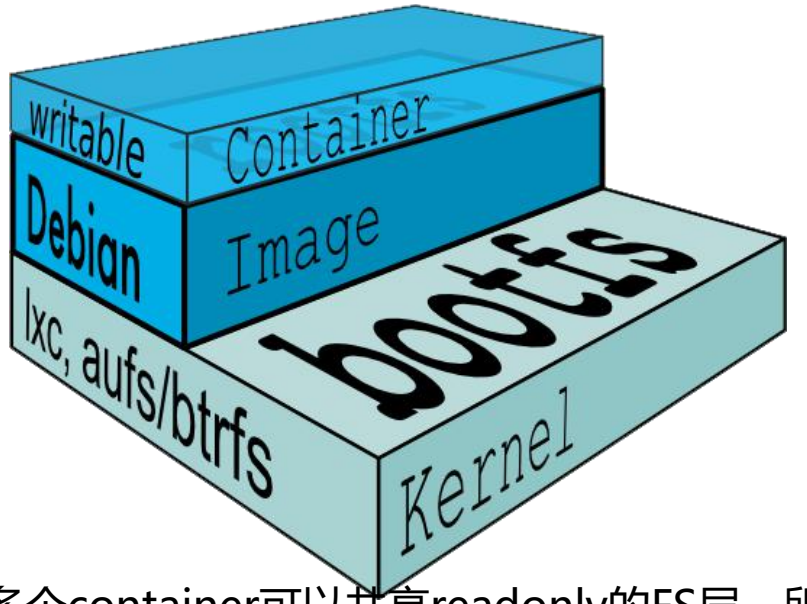
典型的启动Linux运行需要两个FS: bootfs + rootfs:



通常Linux在启动后，首先将 rootfs 设置为 readonly, 进行一系列检查, 然后将其切换为 "readwrite" 供用户使用

Docker核心技术-AUFS

在Docker container初始化时也是将 rootfs 以readonly方式加载并检查，然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的rootfs之上，并且允许再次将下层的 FS(file system) 设定为readonly 并且向上叠加, 这样一组readonly和一个writeable的结构构成一个container的运行时态, 如下图:



这样由于不存在竞争, 多个container可以共享readonly的FS层。所以Docker将readonly的FS层称作 "image" - 对于container而言整个rootfs都是read-write的，但事实上所有的修改都写入最上层的writeable层中, image不保存用户状态，只用于模板、新建和复制使用。

Docker安全策略

——Daemon Attack Surface

——Linux Kernel Capabilities

——Seccomp

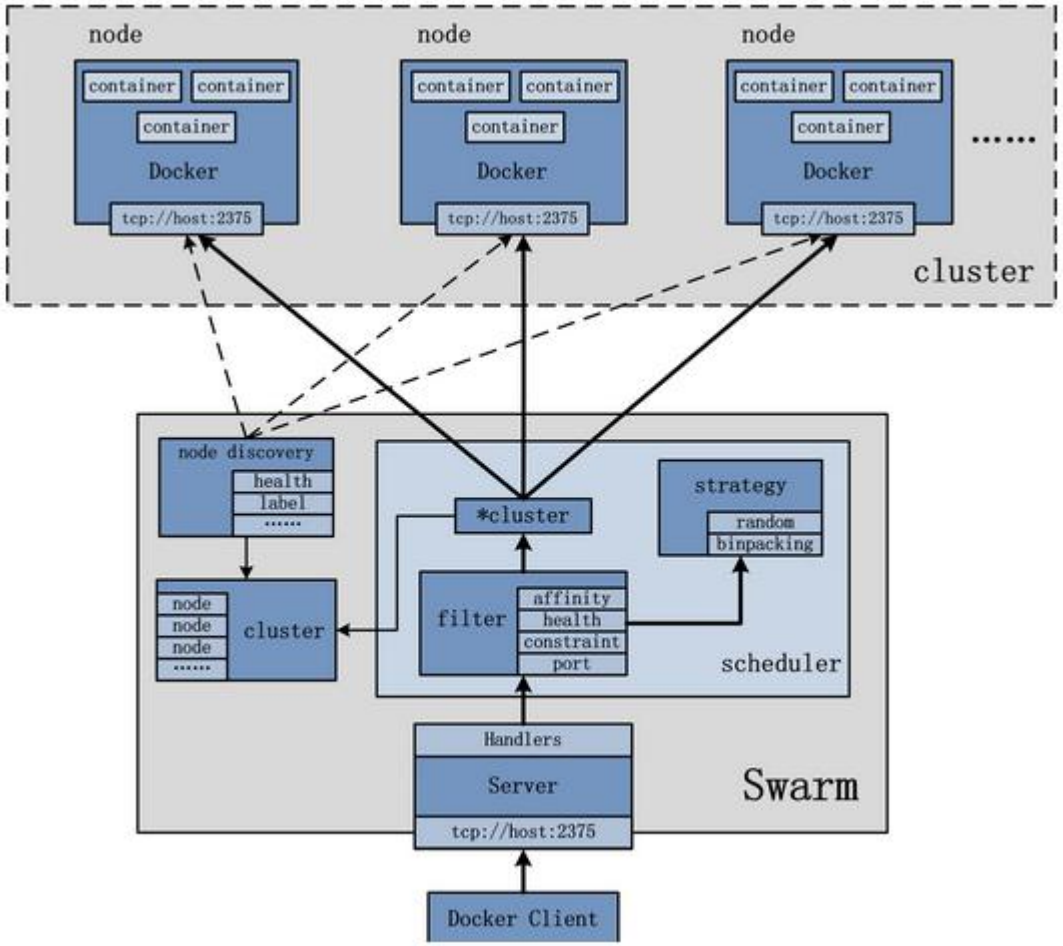
Docker安全策略-Daemon Attack Surface

运行一个容器的核心是通过 Docker Daemon。Docker 服务的运行目前需要 root 权限，因此其安全性十分关键

Docker 允许用户在host和container间共享文件夹，同时不需要限制容器的访问权限，这就容易让容器突破资源限制。例如，恶意用户启动容器的时候将主机的根目录/映射到容器的 /host 目录中，那么容器理论上就可以对host的文件系统进行任意修改了

2016年5月份曝光的Docker Swarm集群管理配置问题，就是将Docker Daemon暴露在公网上，导致任何人都可以使用Docker API来操纵Docker。

Docker安全策略-Daemon Attack Surface



ExecStart=/usr/bin/docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

Docker安全策略

——Daemon Attack Surface

——Linux Kernel Capabilities

——Seccomp

Docker安全策略-Linux Kernel Capabilities

权限机制是Linux内核的另一个强大特性，可以提供细粒度的权限访问控制。
比如有个web应用需要绑定1024以下的端口，只需要拥有CAP_NET_BIND_SERVICE即可，而不需要以root权限启动。

默认情况下，Docker在启动容器时会禁用一部分权限，下面是Docker权限的白名单：

```
s.Process.Capabilities = []string{
                "CAP_CHOWN",
                "CAP_DAC_OVERRIDE",
                "CAP_FSETID",
                "CAP_FOWNER",
                "CAP_MKNOD",
                "CAP_NET_RAW",
                "CAP_SETGID",
                "CAP_SETUID",
                "CAP_SETFCAP",
                "CAP_SETPCAP",
                "CAP_NET_BIND_SERVICE",
                "CAP_SYS_CHROOT",
                "CAP_KILL",
                "CAP_AUDIT_WRITE",
}
```

Docker安全策略-Linux Kernel Capabilities

//加载一个模块到内核空间，并执行模块的init函数。
int init_module(void **module_image*, unsigned long *len*, const char **param_values*);

```
//这个系统调用首先进行权限检查
static int may_init_module(void)
{
        if (!capable(CAP_SYS_MODULE) || modules_disabled)
                return -EPERM;

        return 0;
}
```

```
//检查在init_user_ns这个user_namespace中，是否具有CAP_SYS_MODULE权限
bool capable(int cap)
{
        return ns_capable(&init_user_ns, cap);
}
```

Docker安全策略-Linux Kernel Capabilities


```
int cap_capable(const struct cred *cred, struct user_namespace *targ_ns,
                int cap, int audit)
{
        struct user_namespace *ns = targ_ns;

        for (;;) {
                if (ns == cred->user_ns)
                        return cap_raised(cred->cap_effective, cap) ? 0 : -EPERM;

                if (ns == &init_user_ns)
                        return -EPERM;

                if ((ns->parent == cred->user_ns) && uid_eq(ns->owner, cred->euid))
                        return 0;

                ns = ns->parent;
        }

        /* We never get here */
}
```

Docker安全策略-Linux Kernel Capabilities

大部分情况下，Docker container中的root权限并不是真正的root权限，这意味着即使攻击者在容器中取得了 root 权限，也不能获得本地主机的较高权限，能进行的破坏也有限。

使用权限机制对加强 Docker 容器的安全有很多好处。通常，在服务器上会运行一堆需要特权权限的进程，包括有 ssh、cron、syslogd、硬件管理工具模块（例如负载模块）、网络配置工具等等。容器跟这些进程是不同的，因为几乎所有的特权进程都由容器以外的支持系统来进行管理。

——ssh 访问被主机上ssh服务来管理；
——cron 通常应该作为用户进程执行，权限交给使用它服务的应用来处理；
——日志系统可由 Docker 或第三方服务管理；
——硬件管理无关紧要，容器中也就无需执行 udevd 以及类似服务；
——网络管理也都在主机上设置，除非特殊需求，容器不需要对网络进行配置。

Docker安全策略

——Daemon Attack Surface

——Linux Kernel Capabilities

——Seccomp

Docker安全策略-Seccomp

Secure computing 是Linux内核的另一个特性， Docker利用此特性限制容器内应用程序的行为。

Seccomp 需要内核的支持，在内核编译时需要指定CONFIG_SECCOMP，可以用下面指令检查内核是否开启Seccomp

```
$ cat /boot/config-`uname -r` | grep CONFIG_SECCOMP=
CONFIG_SECCOMP=y
```

Docker安全策略-Seccomp

Docker默认禁用44个系统调用，下表是被禁用的系统调用名称及原因：

Syscall	Description
`acct`	Accounting syscall which could let containers disable their own resource limits or process accounting. Also gated by `CAP_SYS_PACCT` .
`add_key`	Prevent containers from using the kernel keyring, which is not namespaced.
`adjtimex`	Similar to `clock_settime` and `settimeofday` , time/date is not namespaced.
`bpf`	Deny loading potentially persistent bpf programs into kernel, already gated by `CAP_SYS_ADMIN` .
`clock_adjtime`	Time/date is not namespaced.
`clock_settime`	Time/date is not namespaced.
`clone`	Deny cloning new namespaces. Also gated by `CAP_SYS_ADMIN` for CLONE_* flags, except `CLONE_USERNS` .
`create_module`	Deny manipulation and functions on kernel modules.
`delete_module`	Deny manipulation and functions on kernel modules. Also gated by `CAP_SYS_MODULE` .
`finit_module`	Deny manipulation and functions on kernel modules. Also gated by `CAP_SYS_MODULE` .
`get_kernel_syms`	Deny retrieval of exported kernel and module symbols.
`get_mempolicy`	Syscall that modifies kernel memory and NUMA settings. Already gated by `CAP_SYS_NICE` .

Docker安全策略-Seccomp

Docker默认的seccomp profile其实是一个白名单，指定了哪些系统调用是被允许的。

```
{
        "defaultAction": "SCMP_ACT_ERRNO",
        "architectures": [
                "SCMP_ARCH_X86_64",
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
        ],
        "syscalls": [
                {
                        "name": "accept",
                        "action": "SCMP_ACT_ALLOW",
                        "args": []
                },
                {
                        "name": "accept4",
                        "action": "SCMP_ACT_ALLOW",
                        "args": []
                },
                ...
        ]
}
```

Docker安全策略-Seccomp

Seccomp在内核态以Berkeley Packet Filter(BPF)形式实现，主要过滤的是用户态传递过来的系统调用号以及系统调用参数。用户态进程可以调用prctl()来设置seccomp:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, prog);
```

关于seccomp的设置，内核提供2种选项:

1.SECCOMP_MODE_STRICT
该选项限制进程只能够使用read(), write(), _exit()以及sigreturn()，其它系统调用会导致内核向进程传递SIGKILL信号来杀死进程。

2.SECCOMP_MODE_FILTER
用户自定义，过滤系统调用或参数

Docker安全策略-Seccomp

```
static const int mode1_syscalls[] = {
__NR_seccomp_read, __NR_seccomp_write, __NR_seccomp_exit, __NR_seccomp_sigreturn,
};

static void __secure_computing_strict(int this_syscall)
{
        const int *syscall_whitelist = mode1_syscalls;

        do {
                if (*syscall_whitelist == this_syscall)
                        return;
        } while (*++syscall_whitelist);

#ifdef SECCOMP_DEBUG
        dump_stack();
#endif
        audit_seccomp(this_syscall, SIGKILL, SECCOMP_RET_KILL);
        do_exit(SIGKILL);
}
```

Docker逃逸

Q & A

谢　谢