云原生后门扫描探索与实现

原创 christa 在酒吧喝牛奶的牛仔 2023-04-01 16:00 发表于湖南

收录于合集
#扫描器 2 #云原生 1

在云原生高速发展的今天，针对云环境的攻击方法也层出不穷。本文讲述了vesta对云原生后门检测的理念和实践。

DaemonSet

Daemonset因为其特点能够在所有node节点上部署一个pod，因此成为了最受欢迎的后门入口。对Pod设置后门也有如下的几种方法:

- Pod command和arg直接反弹shell
  此类型将反弹shell写入pod的command从而达到对每一个node进行持久化控制，具体内容可以参考CDK的backdoor内容，链接如下

> https://github.com/cdk-team/CDK/blob/3c6b0c1648b25b98067dd308eebcb2a3afb9e95e/pkg/exploit/k8s_backdoor_daemonset.go

因此，改持久化方法对pod的command和arg的内容进行输入检查即可。

- 从其他资源引入反弹代码到Pod中运行
  此类型持久化中比较典型的外部资源一个是Secret，Secret拥有其天然特性Base64编码使得运维人员不能在第一时间看出端倪，但是又能够在Pod内部被ValueFrom调用的时候进行自动解码。亦或者从自制的恶意镜像上拉取进行持久化。

- Sidecar
  考虑到上述几个方案都有痕迹的产生，因此可以考虑k8s自带的kube-proxy，自带的privileged也是满足大多数的渗透要求。可以参考k0otkit脚本以及相关的文章，链接如下

> https://github.com/Metarget/k0otkit

检测

在后门检测中，首先需要对各种权限和安全配置进行检测，因为持久化是后门的本质，攻击者需要长期控制宿主机。因此，需要检测特权权限、必要的capabilities以及相关的挂载。其次，需要检测与 Pod 相关的附加命令，因为现在有许多反向代理的命令和各种绕过的 shell，每次都精确检测都是不现实的。因此，采用对特殊符号和关键字的检查，并将其与字符串总数相除得到一个比率值，然后与相应的阈值进行比较，最后生成报告。

后门相关检测有：

- Secret以及ConfigMap的内容以及解码，判断反代命令的同时也要判断是否是可执行文件

- DaemonSet相关镜像的识别
- Pod的权限以及相关配置检测，例如
  `Privileged`，`volumeMounts`，`command`，`args`，`RBAC` 等

在完成上述检查的开发之后，又发现了另外一个问题。要对DaemonSet进行扫描必然扫描Kubernetes内部关键的namespaces，而这些namespaces又有相当一部分的DaemonSet以及DaemonSet启动的Pod会有特别高的权限，如果加上后期各种监控框架，那么检测出来的结果就会特别繁杂，导致运维人员每次会处理大量无关的安全事件，这是非常消耗时间的。

以一个简单的OpenShift架构以及11个node节点为例子，下列扫描出来的结果都是冗余的

4	DaemonSet	name: nvidia-gpu-exporter | namespace: cloudos-ai-prom	images: os-harbor-svc.default.svc.cloudos:443/ helm/ aios-prometheus-nvidiasmi:1.4	critical	Daemonset has set the unsafe pod "nvidia-gpu-exporter-7sc4f".
5		name: glusterfs | namespace: cloudos-storage	images: registry.cluster.local:9999/ helm/ glusterfs:v6.6	critical	Daemonset has set the unsafe pod "glusterfs-68zq8".
6		name: os-storage-daemonset | namespace: default	images: registry.cluster.local:9999/ library/ containermgt-storage/ pause:3.0	critical	Daemonset has set the unsafe pod "os-storage-daemonset-4tgqc".
7		name: sync | namespace: openshift-node	images: docker.io/ openshift/ origin-node:v3.11.0	critical	Daemonset has set the unsafe pod "sync-b68kp".
8		name: ovs | namespace: openshift-sdn	images: docker.io/ openshift/ origin-node:v3.11.0	critical	Daemonset has set the unsafe
9		name: sdn | namespace: openshift-sdn	images: docker.io/ openshift/ origin-node:v3.11.0	critical	Daemonset has set the unsafe pod "sdn-4lv89".

并且正常的 `kube-system` 命名空间中的kube-proxy也会因为priveleged权限出现在运维人员眼中，出现次数一旦多就会导致对该pod关注度慢慢减少，以至于真正出现后门程序的时候会忽略该实例，导致错过最佳处置时间。

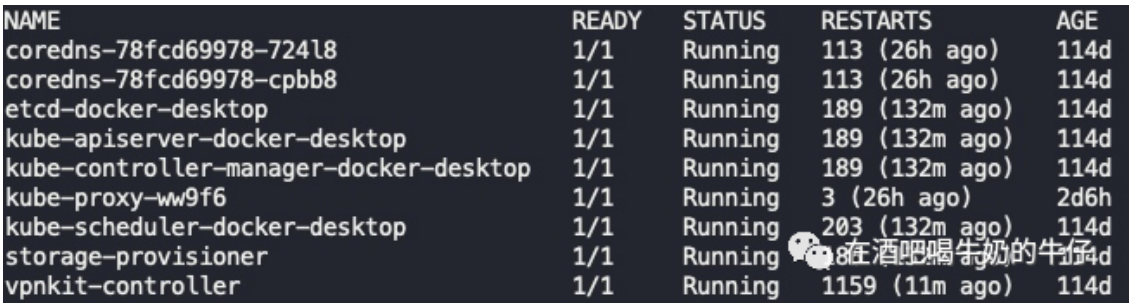
因此vesta对检测的Pod进行了剪枝。我们发现了如下几个参数

- age
- restarts
- certification

age分为node的age以及pod的age，而restarts则是容器重启的次数以及最近一次重启的时间，如图

NAME	READY	STATUS	RESTARTS	AGE
sync-b68kp	1/1	Running	6	1y
sync-d66mf	1/1	Running	4	1y
sync-fptfh	1/1	Running	11	1y
sync-hsvc4	1/1	Running	3	292d
sync-j2cjc	1/1	Running	4	1y
sync-m8sjp	1/1	Running	4	1y
sync-n9zjn	1/1	Running	3	1y
sync-rslr8	1/1	Running	1	61d
sync-s7r28	1/1	Running	3	1y
sync-wt8sq	1/1	Running	5	1y
sync-xjrx4	1/1	Running	3	1y

通过age我们能够直接观察到最近一次修改的时间，通过讲DaemonSet对应的pod中最近修改时间与其他pod的时间进行对比。同时restarts的数值也能够帮助我们一个Pod的正常情况，例如如果一个restarts的次数相对于其他Pod的来说异常地高，可能的行为是反弹的shell所连接的IP并没有连通，导致Pod一直重启尝试连接目标IP，而相对于其他的Pod的平均次数过低的重启次数又能够判断该容器在最近被replace替换过。如下图就是一个非常典型的被加入后门的场景



```
NAME                                     READY   STATUS    RESTARTS         AGE
coredns-78fcd69978-724l8                 1/1     Running   113 (26h ago)    114d
coredns-78fcd69978-cpbb8                 1/1     Running   113 (26h ago)    114d
etcd-docker-desktop                      1/1     Running   189 (132m ago)   114d
kube-apiserver-docker-desktop            1/1     Running   189 (132m ago)   114d
kube-controller-manager-docker-desktop   1/1     Running   189 (132m ago)   114d
kube-proxy-ww9f6                         1/1     Running   3 (26h ago)      2d6h
kube-scheduler-docker-desktop            1/1     Running   203 (132m ago)   114d
storage-provisioner                      1/1     Running   [illegible]      114d
vpnkit-controller                        1/1     Running   1159 (11m ago)   114d
```


而node的age则能够帮助我们获取该node被启动的最早时间，kubernetes安装的证书也同样能够帮助我们拿到K8s最早启动的时间。因此通过目标pod修改时间与node启动最早时间以及其他Pod的修改时间做综合的对比来判断是否跳过该Pod的扫描。

实现的代码如下

- 后门的检测逻辑
  辑: https://github.com/kvesta/vesta/commit/3443fe6e99e17cb40a63b94617a1828092bb4d64#diff-9d440e59ec5a1f856ab57efdf078acf0c0b35adaa036616afa957bec7c48e0b4
- Pod的优化扫描逻
  辑: https://github.com/kvesta/vesta/commit/efaec20c55a3f238516a8187448b91b58a5f822d#diff-e9451bda4fd61e18ef216377ead1083aea189c43f19c0f72c71f016b8f960576R644

至此为vesta对DaemonSet的扫描逻辑以及实现，项目地址为

> https://github.com/kvesta/vesta

收录于合集 #扫描器 2

上一篇 · 镜像扫描Layer分析对比