/proc 攻防演义：从 Linux 进程真相到可观测性实战

/proc Defense in Depth –From Linux Process Truth to Observability

LeisureLinux

2026 年 7 月

目录

0.1 前言：为什么我花了 8 万字写一个”伪文件系统”

某次凌晨三点，运维群里炸了锅。一台生产数据库服务器，CPU 飙到 100%，持续了两个小时。所有监控都亮红灯，但没人能解释是哪个进程——`top` 看到的进程列表完全正常，`ps aux` 也是空的，`/var/log/messages` 干净得像被格式化过。

直到安全工程师登上去，`ls -la /proc/*/exe` 了一下。

谜底揭开：挖矿木马根本**没有创建任何 PID**。它运行在内核线程里，通过劫持合法进程（比如 `kthreadd`）的系统调用，把自己的恶意逻辑嫁接到现有进程的上下文中执行。`/proc` 里看不到它，`ps` 看不到它，但它实际在跑——而且跑得飞起。

但**故事的反转**也来自 `/proc`：因为攻击者忘了清理 `kthreadd` 进程的 `/proc/[pid]/fd/` 目录，安全工程师在 `kthreadd` 的 fd 列表里看到了一个异常 socket，指向一个位于斯洛伐克的 IP。最终顺藤摸瓜，48 小时内完成了清理。

这是 `/proc` 的两面。

它既是**攻击者最爱的藏身地**——因为它记录了进程的一切真相，攻击者必须学会如何在这里”隐身”；它也是**防御者最可靠的眼睛**——因为无论攻击者怎么隐藏，他总要在某个 `/proc` 文件里留下痕迹。

这本书想做的事情，是把 `/proc` 这套接口**当成攻防双方的主战场**来讲。我们不会重复 `man proc` 里那些冰冷的字段说明，也不会照抄 node_exporter 的源码注释。我们要看的是：

- 攻击者**怎么**用 `/proc` 摸清环境、嗅探密钥、隐藏后门
- 防御者**怎么**用 `/proc` 定位攻击、识别异常、构建可观测性体系
- 在容器化、eBPF 化的浪潮里，`/proc` 这套诞生于 1992 年的接口，**下一站**会走向哪里

全书约 4 万中文字，分 7 章，结构上沿用了我们写《SSH 暴力破解深度防御》时的”攻击者视角 + 纵深防御 + 实战案例 + 可观测性落地”骨架。如果您读过那本书，会很快找到熟悉的节奏感；如果没读过，也不影响您从零开始读这本。

写作过程中，我尝试把每一个抽象概念都对应到一段**真实命令输出**或一个**可复现的实验**。我相信安全工程师和 SRE 都不喜欢”正确而无用的废话”——所以这本书的代码块占比会比较高。

如果您读完后，能在下一次告警时多看一眼 `/proc` 下那几个关键文件，能在新服务上线时多考虑一步 `/proc` 暴露面，能在容器逃逸 CVE 公布时第一时间意识到它涉及 `/proc` 哪个路径——这本书就完成了它的使命。

江湖路远，`/proc` 无声。愿这本书能成为您读懂它的一把钥匙。

——LeisureLinux 2026 年 7 月

0.2 第 1 章：/proc 为什么是头号攻击面

0.2.1 数字不会说谎：/proc 暴露的进程真相

打开任意一台运行中的 Linux 服务器，执行一句命令：

```
ls /proc | head -20
```

你看到的不是”几个文件”，而是这台机器**此时此刻的全部真相**：

- 每秒变化的心跳（`/proc/loadavg`、`/proc/uptime`）
- 全部运行的进程（`/proc/[pid]/`）
- 全部打开的文件、socket、管道（`/proc/[pid]/fd/`）
- 全部的网络连接（`/proc/net/tcp`、`/proc/net/tcp6`）
- 全部的内存映射（`/proc/[pid]/maps`）
- 全部的环境变量（`/proc/[pid]/environ`）

`/proc` 不是普通的文件系统。它是 **Linux 内核的自我披露接口**——内核用一种叫做 procfs 的虚拟文件系统，把运行时的一切状态”打印”在 `/proc` 目录下，供用户态查询。

这个设计带来一个奇妙的副作用：

> **只要你能读到 /proc，你就能完整地”看见”一台 Linux 机器。**

这正是它成为头号攻击面的根本原因。

0.2.1.1 一个真实的端口排查场景

假设生产上发现某个端口被占用了，普通工程师会这么查：

```
ss -ltnp | grep :8080
```

老一点的工程师会：

```
netstat -ltnp | grep :8080
```

资深一点的工程师会直接：

```
ls -la /proc/*/fd/* 2>/dev/null | grep socket
# 找到 socket inode
cat /proc/net/tcp | grep "<inode>"
# 反查到 PID
```

这条 `ls /proc/*/fd/*` 命令，**几乎是所有端口排查的”终极武器”**——因为无论攻击者怎么改 `ss`、改 `netstat`、改 `lsof`，他都改不了内核在 `/proc/[pid]/fd` 里记录的真实 fd 链表。这是 `/proc` 在运维侧的”不可替代性”。

但同样这条命令，**也是攻击者最喜欢的侦察起点**。原因很简单：他能从这里摸到所有进程的 fd，能顺着 socket 找到所有网络连接，能顺着可执行文件找到所有程序，能顺着 cwd 找到所有工作目录，能顺着 environ 找到所有密钥。

这就是 `/proc` 的双重身份：**它既是运维的明灯，也是攻击者的情报源**。

0.2.1.2 `/proc` 在安全事件中的”出席率”

2024 年到 2026 年，是 Linux 内核历史上 CVE 披露最密集的时期之一。仅这两年间，直接命中 `/proc` 的高危漏洞就至少包括：

- **CVE-2025-31133 / 52565 / 52881（runc 三连击）**：影响 Docker / containerd / **Kubernetes** / **Podman** 全系容器运行时，可在容器启动期通过 `/proc/sys/kernel/core_pattern` 与 `/proc/sysrq-trigger` 实现宿主级 root 执行
- **CVE-2025-40271**：直接命中 `/proc` 文件系统的内核函数 `proc_readdir_de()` 的 use-after-free，理论上可导致本地提权或信息泄露
- **CVE-2026-46333**：基于 `get_dumpable()` race condition，让普通用户读 SSH host 私钥、`/etc/shadow` 等敏感文件
- **CVE-2022-0847 (DirtyPipe)** 与 **CVE-2026-31431 (Copy Fail)**：虽然不直接走 `/proc` 路径，但前者通过 `/proc/self/mem` 完成只读文件改写，后者影响 2017 年以来的所有主流发行版——它们揭示了同一个规律：**只要涉及内存、进程、文件缓存的漏洞，最终都会找到 `/proc` 这条路径**

把时间拉得更长：pkexec（PwnKit, 2021）的 `/proc/self/environ` 注入、DirtyPipe（2022）的 `/proc/self/mem` 改写只读文件、ns_last_pid（2022）的 namespace 逃逸、cgroups v1 release_agent（2022）的容器逃逸、OverlayFS（2023）的 `/proc/[pid]/root` 绕过——这些 CVE 都不只是”某个文件被滥用”，它们构成了一**条贯穿 Linux 权限边界演化的攻击链**。

所以，写一本关于 `/proc` 的书，本质上是在**盘点 Linux 内核最容易被攻击的那一面**——而且这一面每隔几个月就会多几个新的入口。

0.2.2 `/proc` 的本质：进程即目录，内核即文件系统

要真正理解 `/proc` 的攻击面与可观测性价值，我们必须先理解它的设计哲学。

0.2.2.1 procfs 的诞生

1992 年，Linux 内核 0.99 引入了一个叫 procfs 的虚拟文件系统。它的设计初衷极其朴素：

> 让用户态进程能通过”读文件”的方式查询内核状态。

在当时，这只是一个方便 `ps` 命令读取进程信息的辅助功能。但这个设计被证明**太有用了**——后来几乎所有”暴露内核状态”的需求都通过 procfs 实现。到 Linux 2.6 时代，`/proc` 已经膨胀到上百个文件；到 Linux 5.x，`/proc` 已经有 200+ 文件，跨越 7 大子系统。

0.2.2.2 procfs 在内核 VFS 中的位置

/proc 不是块设备、不是网络文件系统，它是 **VFS（Virtual File System）层的一种特殊挂载**。它的特点：

- 文件**不存储在磁盘上**——它们是内核数据结构的”视图”
- 读操作触发**内核函数**（read 回调）
- 写操作触发**内核函数**（write 回调）
- 列出目录（ls）触发**内核函数**（iterate 回调）

这意味着 /proc 下的每一个文件，背后都对应一**段内核代码**：

```
// 例如 /proc/[pid]/status 的读操作
static int proc_pid_status(struct seq_file *m, struct pid_namespace *ns,
                           struct pid *pid, struct task_struct *task) {
    struct mm_struct *mm = get_task_mm(task);
    ...
    seq_printf(m, "Name:\t%s\n", task->comm);
    seq_printf(m, "Pid:\t%d\n", pid_nr_ns(pid, ns));
    ...
}
```

这段代码**每次你 cat /proc/[pid]/status 时都会被执行**——/proc 文件不是静态的”快照”，而是**每次访问都实时生成的内核状态视图**。

0.2.2.3 “进程即目录”的统一范式

procfs 最天才的设计是把 **PID 当作目录名**：

```
/proc
├── 1/          ← PID 1（通常是 init）
│   ├── cmdline
│   ├── environ
│   ├── status
│   ├── maps
│   ├── fd/
│   │   ├── 0 → /dev/null
│   │   ├── 1 → /dev/null
│   │   └── 4 → socket:[12345]
│   └── ...
├── 2/          ← PID 2（通常是 kthreadd）
├── 1234/       ← 普通进程
└── ...
```

这个范式让”查询进程状态”变成**普通文件系统操作**——你不需要特殊的 API，直接 cat、ls、grep 就行。这种”统一性”让 /proc 在运维和攻击两个维度上都成为首选入口。

0.2.2.4 /proc 与 sysfs 的分工

你可能注意到，Linux 还有另一个”伪文件系统”——/sys（sysfs）。它和 /proc 经常被混淆，但分工完全不同：

维度	/proc (procfs)	/sys (sysfs)
设计年代	1992（Linux 0.99）	2003（Linux 2.6）
主要用途	暴露**进程**和**内核状态**	暴露**设备**和**驱动**
文件数量	~200	~10000+
写操作	谨慎使用	经常使用（设备配置）
安全敏感度	**极高**（进程、内存、网络）	中等（设备配置）
可观测性价值	**极高**（运行时全貌）	中等（硬件视角）

简单说：**/proc 关注”软件运行时”，/sys 关注”硬件配置时”**。本书专门讲 /proc，但会涉及 /proc/sys/ 这个交叉地带——它是 procfs 的子集，却承担了 sysctl 功能的实现。

0.2.3 真实事故：那些年，/proc 见证的安全灾难

让我们回到 2024-2026 年的真实世界，看看 /proc 是怎么在安全事件中扮演关键角色的。这一节选出的四个案例，是当时**对生产环境影响最大、/proc 因素最清晰、修复成本最高的**四类事故。

0.2.3.1 runc 三连击（2025 年 11 月）：容器时代的”新 DirtyPipe”

时间线：2025 年 11 月公开披露，三个 CVE 同步公布：- **CVE-2025-31133**（maskedPaths 滥用）- **CVE-2025-52565**（/dev/console mount 竞态）- **CVE-2025-52881**（任意 procfs 写入重定向）

影响范围：runc 1.0.0-rc3 至 1.4.0-rc.2 全系，几乎所有使用 Docker、containerd、**Kubernetes（kubelet 默认依赖 containerd/shim）**、**Podman** 的生产集群都受影响。修复版本是 1.2.8 / 1.3.3 / 1.4.0-rc.3。

漏洞原理（一句话版）：runc 在容器启动时以**宿主 root 身份**在宿主的 procfs 上执行 mount 操作，但**没有充分验证挂载源的身份**——攻击者可以在挂载动作发生的瞬间，用一个符号链接替换原本应是 /dev/null 的目标，让 runc “贴心”地把宿主 /proc 下任意敏感文件**以读写模式 bind-mount 进容器**。

为什么对 K8s/Podman 用户特别致命：在 K8s 集群里，kubelet 默认使用 containerd + runc 来拉镜像并启动 Pod。如果攻击者能够**控制 Pod spec 里的 mount 配置**（比如通过提交一个带恶意 volumeMount 的 manifest），就能触发这三个 CVE 中的任何一个。Podman 的 rootful 模式同样在 runc 之上，触发条件一致。

攻击链路（以 CVE-2025-31133 为例）：

```
# 1. 攻击者构建一个含恶意 entrypoint 的镜像
#    entrypoint.sh 做了三件事：
#    a) 在容器内的 /dev/null 上做准备工作
#    b) 在 runc 执行 maskedPaths 挂载前，把 /dev/null 替换成符号链接
#       指向宿主 /proc/sys/kernel/core_pattern
#    c) 等待 runc 完成 bind-mount

# 2. runc 执行 bind-mount 时：
#    它认为自己在挂载 /dev/null（写保护）
#    实际上挂的是宿主 /proc/sys/kernel/core_pattern（读写）

# 3. 攻击者在容器内写入 core_pattern：
echo "|/proc/self/exe" > /proc/sys/kernel/core_pattern
#    core_pattern 写成 "| 程序路径" 时，内核执行该程序以 root 身份处理 core dump
#    攻击者让自己的进程 crash，内核以 root 身份执行恶意程序
#    → 容器逃逸成功，宿主机 root
```

/proc/sys/kernel/core_pattern 在这里的角色是**内核的”程序执行器”**——它本来用于”core dump 时调用哪个程序处理 dump 文件”，但被当成了”以 root 执行任意命令的后门”。这与 CVE-2025-52881 滥用的 /proc/sysrq-trigger（写一个字符即可触发系统级动作）形成一组**“/proc/sys 即武器库”**的当代攻击范式。

防御侧的教训：

- **必须升级 runc**：1.2.8 / 1.3.3 / 1.4.0-rc.3 及以上
- **启用 user namespace**：让容器内 root ≠ 宿主 root
- **拒绝不受信镜像**：恶意镜像的 entrypoint 是触发条件
- **Kubernetes 侧**：在 PodSpec 里设置 securityContext.allowPrivilegeEscalation: false,并限制 hostPath 类型 mount
- **Podman 侧**：用 --userns=keep-id 或 rootless 模式

这个 CVE 系列印证了一个观点：**/proc 不只是只读接口，它是容器运行时对宿主内核的”操作通道”**——任何对这个通道的写入放大，都会立刻变成容器逃逸能力。

0.2.3.2 CVE-2026-46333：通过 get_dumpable() race condition 读 SSH host 私钥

时间线：2026 年初披露。漏洞自 Linux 4.10-rc1（2016 年 11 月）起就存在，**潜伏十年**。

漏洞原理：ptrace_may_access() 调用 get_dumpable() 检查目标进程是否允许被 ptrace。但 get_dumpable() 读的是 task_struct 的一个字段，**进程退出时这个字段的更新存在一个时间窗口**——攻击者在这个窗口内 attach 一个特权进程（如 sshd、cron job），就能读取该进程的内存。

/proc 在这里扮演什么角色：

- 攻击链路里，攻击者通过 /proc/[pid]/status、/proc/[pid]/stat、/proc/[pid]/auxv **实时探测特权进程的退出时机**
- 一旦窗口打开，立即通过 ptrace attach 上去读内存
- SSH host 私钥、/etc/shadow、dbus-daemon 的凭证都可能在这一刻被读出

为什么这件事对 /proc 重要：

这是”/proc 作为攻击者探测仪表盘”的典型——**仅仅靠读取 /proc 就能识别出特权进程在什么时候退、什么时候进入 race window**，整个攻击的”难度评估”完全建立在 /proc 提供的实时可见性上。

防御侧的教训：

- 升级内核
- 设置 /proc/sys/kernel/yama/ptrace_scope = 1（仅允许同用户进程 ptrace）或更高
- 在容器里设置 allowPrivilegeEscalation: false 和 SCC 限制

0.2.3.3 CVE-2025-40271：proc_readdir_de() use-after-free

时间线：2025 年披露。Red Hat 已发布 RHSA-2026:3360 等多个发行版补丁。

漏洞原理：proc_readdir_de() 在遍历 /proc 目录时，使用红黑树（rbtree）管理子节点。当一个 proc_dir_entry 被 rb_erase() 从树中移除时，**对应的节点没有被显式置为 EMPTY**，这给后续的 pde_subdir_next() 留下了一个 use-after-free 窗口。

触发路径：攻击者需要 CAP_NET_ADMIN 权限（可创建/删除网络设备）。攻击者一边遍历某个特定目录（/proc/pid/net/dev_snmp6/ 等），一边反复创建/删除 tun 设备，制造竞态。

/proc 在这里不是攻击者工具，而是攻击面——它揭示了 **procfs 自身实现 ** 的脆弱性，不依赖任何用户态程序。

防御侧的教训：

- 升级内核（补丁是给 rb_erase() 后加 RB_CLEAR_NODE()）
- 限制 CAP_NET_ADMIN 的发放
- 在容器里完全 drop CAP_NET_ADMIN（除非业务必须）

0.2.3.4 挖矿木马：通过 /proc/net/tcp 隐藏 C2

时间线：2021-2026 年多个挖矿木马家族（包括 Kinsing、TeamTNT、WatchBog）的常规操作。

攻击手法：

```
# 攻击者拿到初始访问后：
1. 修改 iptables 规则，把端口 22/80/443 之外的所有出站流量 DROP
2. 修改 /etc/ld.so.preload 加载恶意库，hook 掉 ss/netstat/lsof
3. 把 ss/netstat/lsof 替换成只显示白名单连接的版本
4. 通过 /proc/net/tcp 直接连出（绕过 ss 的 hook）
```

/proc/net/tcp 在这里成为攻击者的**“备用通道”**——当用户态工具都被 hook 时，/proc 作为内核态接口仍然可用。

防御侧的教训：

```
# 防御侧永远要交叉验证
ss -tnp      # 看到连接 A
cat /proc/net/tcp  # 看到连接 B（ss 漏的）
```

这就是为什么我们说 /proc 是**最后的事实来源**——任何用户态工具都可以被 hook，但内核直接生成的 /proc 数据难以伪造。

0.2.3.5 时代的对照：CVE-2022-0847 与 CVE-2026-31431

最后，把目光放得更远一点。**2022 年的 DirtyPipe（CVE-2022-0847）**攻击的是 /proc/self/mem，允许覆盖任意只读文件；**2026 年的 Copy Fail（CVE-2026-31431）**攻击的是 AF_ALG 接口的 algif_aead 模块，让未授权用户能破坏 setuid 二进制的 page cache。这两个 CVE 看似无关，但它们揭示了同一个事实——

> **/proc 暴露的内存视图与内核里的真实缓存之间，是攻击者最常用的”借道”路径。**

这也意味着本书不是”对一个老话题的重述”，而是在一个**仍在持续暴露新入口的领域**里做系统盘点。

0.2.4 安全本质：/proc 是便利与失控的拉锯

到这里，你应该已经感受到 /proc 的两面性。

便利的一面：它让 Linux 的可观测性变得极其简单——任何工程师都可以用 cat、ls、grep 这些基本命令，看到系统的一切。

失控的一面：正是这种”一切可见”的便利，让攻击者也获得了同样的能力。

这就是我们写这本书的根本动机：**/proc 的便利和失控是一体两面的**。我们不能为了安全而牺牲可观测性（那会让我们失去眼睛），也不能为了便利而忽视暴露面（那会让攻击者拿走情报）。

正确的姿势是：

> **理解 /proc 的每一个文件的语义和风险，建立纵深防御，把暴露面降到合理水平，同时保留必要的可观测性**。

0.2.4.1 2026 年的启示

本书写作的 2026 年，正值两个现实：

- **runc 三连击**（CVE-2025-31133/52565/52881）的余波还在——几乎所有 K8s/Podman 集群都被迫紧急升级 runc，1.2.8 / 1.3.3 / 1.4.0-rc.3 之前的版本从那一刻起都不再被信任
- **Copy Fail**（CVE-2026-31431）让 2017 年以来所有发行版紧急打补丁，原因是内核中的 in-place 优化被错误地保留了近十年

这两个 CVE 都不是”/proc 路径”上的漏洞，但它们的**修复、缓解、告警**，无一例外都依赖 /proc 提供的事实：升级 runc 时需要确认容器进程的 /proc/[pid]/comm、/proc/[pid]/exe 路径正确；检测 Copy Fail 利用尝试时需要监控 /proc/[pid]/maps 中异常的可执行文件映射；审计容器逃逸时需要从 /proc/net/tcp 还原 C2 连接。

> **/proc 作为 Linux 内核最古老的攻击面，34 年来从未真正”收敛”过**——它的演化与 Linux 内核的演化同频共振。

具体怎么做？从第 2 章开始，我们进入攻击者视角，看他们怎么用 /proc 摸清机器、嗅探密钥、隐藏后门。然后在第 3 章，我们转向防御者视角，搭建 5 层纵深防御体系。

> **下一章预告**：第 2 章 **攻击者怎么用 /proc** ——我们会从环境指纹开始，沿着”摸环境 → 找进程 → 读内存 → 看网络 → 隐身形 → 跨边界”六步，把攻击者的 TTP（Tactics, Techniques, Procedures）逐一拆解。每一个 TTP 都会配上具体的命令、真实的输出、可能的对抗手段。

0.3 第 2 章：攻击者怎么用 /proc

0.3.1 攻击者视角的总览

在我们深入具体 TTP 之前，先把攻击者的”作战图”画出来。

一个拿到任意 Linux 机器最低权限（甚至只是一个 web shell）的攻击者，要做下一件事——**判断这台机器值不值得深度打**。这不需要任何漏洞利用，只需一组 /proc 命令：

```
# 30 秒环境评估
uname -a                              # 内核版本（决定有哪些 CVE 可用）
cat /proc/cpuinfo | grep -c "^processor"  # CPU 核数（决定挖矿收益）
cat /proc/meminfo | head -3           # 内存总量
df -h                                 # 磁盘（决定能不能塞大文件）
cat /proc/net/route                   # 网络（有没有内网）
cat /proc/1/cgroup                    # 是否在容器内
cat /etc/os-release                   # 发行版（决定怎么提权）
```

这 7 个动作，前 5 个走的是 /proc 接口。如果答案是「内核老 + CPU 多 + 在容器内 + 有内网」，攻击者就会下决心深打——**因为这种机器往往不是终点，而是跳板**。

这一章我们要还原的，是深打开始后，攻击者**沿着 /proc 一路向里走**的全过程。我把它拆成六步：

步	任务	主要 /proc 文件
1	摸环境：环境指纹 + 内核探测	/proc/version、/proc/cpuinfo、/proc/config.gz
2	找进程：枚举目标进程 + 嗅探密钥	/proc/[pid]/cmdline、/proc/[pid]/environ
3	读内存：进程内存取证 + 注入	/proc/[pid]/maps、/proc/[pid]/mem
4	看网络：连接发现 + C2 通道	/proc/net/tcp*、/proc/net/arp
5	隐身形：rootkit 与 /proc 的对峙	/proc/kallsyms、/proc/[pid]/fd
6	跨边界：容器与 namespace 越界	/proc/1/root、/proc/sys/kernel/ns_last_pid

每一步，我们都会从攻击者的角度给出”具体怎么做”，再从防御者的角度给出”在哪里能看见他”。这是 /proc 攻防的核心节奏：**它既给攻击者递刀，也给防御者递盾**。

0.3.2 第一步：摸环境——环境指纹与内核探测

0.3.2.1 攻击者想要的 7 项情报

任何拿到 shell 的攻击者，第一件事不是提权，而是**判断机器画像**。一个没有目标的提权是浪费，一个有目标的提权价值翻倍。/proc 提供以下 7 项关键情报：

```
# 1. 内核版本（决定可利用 CVE）
cat /proc/version
# 典型输出：Linux version 5.15.0-91-generic (buildd@lcy02-amd64-010) ...

# 2. CPU 信息（决定挖矿程序选型与多线程加速）
cat /proc/cpuinfo | head -25
# 关键字段：model name、cpu MHz、cache size、flags（包括 vmx/svm 决定能不能跑虚拟机）

# 3. 内存（决定挖矿和 DoS 门槛）
cat /proc/meminfo | head -5
# 关键字段：MemTotal、MemAvailable、SwapTotal

# 4. 启动时长（判断机器是新部署还是生产老机器）
cat /proc/uptime
# 输出：12345.67 6789.01（运行秒数 + 空闲秒数）

# 5. 平均负载（判断机器是否空闲，是否适合长期潜伏）
cat /proc/loadavg
# 输出：0.12 0.34 0.56 1/234 5678

# 6. 网络路由（决定有没有内网可扫）
cat /proc/net/route
# 输出表格式：Iface | Destination | Gateway | Flags | RefCnt | Use | Metric | MTU | Window | IRTT

# 7. 是否在容器内（决定攻击路径是不是只到容器边界）
cat /proc/1/cgroup
# 容器内会有 12:devices:/docker/<id> 或 0::/kubepods/... 这种路径
```

把这 7 项叠在一起，攻击者几秒钟就能画出机器画像。**/proc 在这里充当的是”机器身份证”——它比 uname -a 详细得多，比 /etc/os-release 实时得多，比任何主动扫描都隐蔽**。

0.3.2.2 通过 /proc/config.gz 识别隐藏的内核特性

多数发行版的内核是模块化编译的——某项安全特性有没有启用，对应着完全不同的攻击路径。攻击者可以用 /proc/config.gz（如果挂载了）反推内核编译选项：

```
# 检查是否有 config.gz
ls /proc/config.gz 2>/dev/null
zcat /proc/config.gz | grep -E "CONFIG_(SECCOMP|LSM|YAMA|USERLANDER)"
# 关键输出：
# CONFIG_SECCOMP=y
# CONFIG_SECURITY=y
# CONFIG_SECURITY_APPARMOR=y
# CONFIG_SEC_YAMA=y
# CONFIG_USER_NS=y
```

这些配置项决定了后续攻击路径的可走性：

- **CONFIG_SECCOMP** 没编进内核 → seccomp 不可用，第 3 章很多防御规则失效
- **CONFIG_USER_NS** 编进了内核 → 用户 namespace 可用，容器逃逸路径更多
- **CONFIG_SEC_YAMA** 没编 → ptrace_scope 无法设置，CVE-2026-46333 这类 ptrace race 完全可利用

防御侧的对照：

```
# 我们需要知道暴露面有多大
zcat /proc/config.gz | grep -E "CONFIG_(KASLR|HARDENED|STATIC_USERMODEHELPER|RANDOMIZE)"
# 输出：
# CONFIG_RANDOMIZE_BASE=y    # KASLR
# CONFIG_STATIC_USERMODEHELPER=y  # 内核调用 usermodehelper 的限制
```

0.3.2.3 内核模块探测：哪些”可装载驱动”会被攻击者偏好

```
# 当前加载的内核模块
cat /proc/modules
# 输出格式：模块名 大小 引用计数 依赖列表 [状态] 地址

# 用 modinfo 可以看每个模块的描述（但需要 root）
lsmod
```

攻击者会重点关注两类模块：

- **网络相关**：tcp_bbr、br_netfilter、nf_conntrack、tun（CVE-2025-40271 的触发条件之一）
- **安全相关**：apparmor、selinux、landlock

tun 设备特别值得警惕——很多 CVE（不止 CVE-2025-40271）都通过反复创建/删除 tun 设备来制造 /proc 路径上的竞态。

0.3.3 第二步：找进程——枚举目标进程与嗅探密钥

0.3.3.1 进程指纹五件套

找到目标进程是攻击者下一步动作。/proc/[pid]/ 下有 5 个文件构成”进程指纹五件套”：

```
# 在 ps/top 之外，用 /proc 直接抓进程
for pid in /proc/[0-9]*; do
  pid_num=$(basename $pid)
  echo "=== PID $pid_num ==="
  echo "  cmdline: $(tr '\0' ' ' < $pid/cmdline)"
  echo "  comm:    $(cat $pid/comm)"
  echo "  exe:     $(readlink $pid/exe)"
  echo "  cwd:     $(readlink $pid/cwd)"
  echo "  user:    $(grep Uid $pid/status | awk '{print $2}')"
done
```

这 5 个文件对应了”**进程是什么、跑在哪、用什么身份**”的完整画像：

- cmdline：启动命令（容易透露数据库连接串、密钥文件路径）
- comm：进程短名（16 字符以内，可能被攻击者改写）
- exe：可执行文件（攻击者可借此定位 SO 加载路径）
- cwd：工作目录（很多程序会在这里放临时文件）

- status：Uid 行是真实用户身份（与 euid 可能不同——这一点很关键，SUID 程序会暴露）

0.3.3.2 /proc/[pid]/environ 的密钥泄露

环境变量是攻击者的金矿。在云原生时代，**大量密钥直接通过环境变量注入**：

```
# 嗅探某进程的环境变量
cat /proc/$(pidof mysqld)/environ | tr '\0' '\n'
# 典型输出：
# PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
# MYSQL_ROOT_PASSWORD=ProductionP@ssw0rd
# AWS_ACCESS_KEY_ID=AKIA...
# AWS_SECRET_ACCESS_KEY=wJalr...
# DATABASE_URL=postgres://admin:secret@db.internal:5432/app
```

权限问题：**environ 文件的读权限是进程创建时决定的**。如果进程以 root 启动（如常见的 systemd-managed 服务），/proc/[root_pid]/environ 对所有本地用户可读。**这是 PwnKit（CVE-2021-4034）攻击链路的前置条件**。

0.3.3.3 攻击者的”高价值进程清单”

拿到 SSH shell 后，攻击者会优先找这几类进程：

```
# 一键找出所有含 "key/token/secret/password" 的环境变量
for pid in /proc/[0-9]*; do
  if [ -r $pid/environ ]; then
    matches=$(cat $pid/environ 2>/dev/null | tr '\0' '\n' | grep -iE "(KEY|TOKEN|SECRET|PASSWORD|CRED)")
    if [ -n "$matches" ]; then
      echo "PID $(basename $pid): $matches"
    fi
  fi
done
# 这条命令一发，能跑出全机的明文凭证清单
```

这是 /proc 的双面性最尖锐的体现：**同样的命令，运维用来排查泄漏，攻击者用来收集密钥**。

0.3.3.4 防御侧的”进程嗅探防护”

```
# 1. 用 hidepid 限制其他用户读 /proc/[pid]/environ
mount -o remount,hidepid=2 /proc
# 隐藏 PID 后，普通用户看不到其他 PID 的目录

# 2. 让服务进程用专用低权限用户运行（让 environ 只能被该用户读）
# 例如 systemd 的 DynamicUser=yes 或 User=mysql

# 3. 用 secret manager 替代环境变量（HashiCorp Vault、AWS Secrets Manager）
#    让进程在启动时拉密钥，而不是通过环境变量注入
```

0.3.4 第三步：读内存——进程内存取证与注入

0.3.4.1 /proc/[pid]/maps：进程的”完整地图”

maps 是攻击者必看的文件。它显示进程的全部内存映射——包括共享库、可执行文件、堆、栈：

```
cat /proc/$(pidof nginx)/maps | head -20
# 输出样例：
# 00400000-00401000 r-xp 00000000 fd:01 12345  /usr/sbin/nginx
# 00600000-00601000 r---p 00000000 fd:01 12345  /usr/sbin/nginx
# 00601000-00602000 rw-p 00001000 fd:01 12345  /usr/sbin/nginx
# 7f1234000000-7f1234200000 r-xp 00000000 fd:01 67890  /usr/lib/x86_64-linux-gnu/libssl.so.3
# 7fffabcd0000-7fffabcdf000 rw-p 00000000 00:00 0   [stack]
# ...
```

从攻击者视角，maps 揭示了：

- **可执行文件的加载基址** → 用于定位 GOT/PLT → 实施 GOT 覆盖攻击
- **共享库的版本** → 对应已知 CVE（如 libssl.so.3 对应 CVE-2024-xxxx）
- **堆栈范围** → 用于栈溢出布局
- **匿名映射** → 隐藏的代码注入区域

0.3.4.2 /proc/[pid]/mem：直接读/写进程内存

mem 文件按地址读，配合 maps 可以把进程的整个内存 dump 出来：

```
# 用 gcore 完整 dump（最常用工具）
gcore $(pidof mysqld)
# 生成 core.<pid> 文件，里面有进程的全部内存

# 手动版本：从 maps 提取所有可读段，然后 lseek+read
gdb -p $(pidof mysqld) -batch -ex "dump memory /tmp/mysql.mem 0x400000 0x401000"
# 注意：dump memory 是 gdb 的内存 dump 命令
```

攻击者最爱的用法：

1. gcore <pid> 抓核心 dump
2. 用 strings、grep 在 dump 里搜密钥、密码、token
3. 对 dump 做行为分析（解密内存中的 TLS 会话密钥）

0.3.4.3 mem 文件的写权限：内存注入的入口

/proc/[pid]/mem 不仅可读，**还可写**。在合适的权限下，攻击者可以直接向进程内存注入代码：

```
# 写内存（需要 ptrace 权限，或与目标进程同 uid）
# 简化示例：往进程的代码段注入 shellcode
gdb -p $(pidof victim) -batch -ex "set *(unsigned char*)0x400000 = 0x90"  # NOP 滑板

# 实际攻击中，攻击者会：
# 1. 关闭 ASLR（如果可能）
# 2. 在堆上分配 RWX 内存
# 3. 写入 shellcode
# 4. 改函数指针（GOT/PLT）指向 shellcode
```

DirtyPipe（CVE-2022-0847）正是利用了 /proc/self/mem 的写能力完成只读文件改写。

0.3.4.4 防御侧的”内存边界保护”

```
# 1. kernel.yama.ptrace_scope：限制 ptrace 范围
echo 1 > /proc/sys/kernel/yama/ptrace_scope
# 0 - classic（任何同 uid 进程都可 attach）
# 1 - restricted（仅父子进程）
# 2 - admin only（仅 CAP_SYS_PTRACE）
# 3 - no ptrace（完全禁用）

# 2. 内存 dump 限制（仅 root 可执行）
# 默认就是 root only，普通用户即使能读 /proc/[pid]/maps 也无法 mem read

# 3. 内核硬化的开关
cat /proc/sys/kernel/randomize_va_space  # 2 = 完整 ASLR
cat /proc/sys/kernel/kptr_restrict       # 2 = 隐藏内核符号
cat /proc/sys/kernel/dmesg_restrict      # 1 = 仅 root 可读 dmesg
```

0.3.5 第四步：看网络——连接发现与 C2 通道

0.3.5.1 /proc/net/ 是 C2 通道的事实来源

/proc/net/ 下有 20+ 个文件，每个都对应一种网络栈视图。攻击者最关心 6 个：

```
# 1. 当前所有 TCP 连接（v4）
cat /proc/net/tcp
# 字段: sl | local_address | rem_address | st | tx_queue | rx_queue | ...
# st = 01 = ESTABLISHED
# local_address 和 rem_address 是 hex-encoded:port

# 2. 当前所有 TCP 连接（v6）
cat /proc/net/tcp6

# 3. UDP socket
cat /proc/net/udp

# 4. ARP 表（内网拓扑）
cat /proc/net/arp

# 5. 路由表
cat /proc/net/route

# 6. 设备统计
cat /proc/net/dev
```

地址是十六进制编码的——比如 `0100007F:1F90` 表示 `127.0.0.1:8080`（注意字节序：01 00 00 7F 倒过来就是 7F 00 00 01 = 127.0.0.1）。

0.3.5.2 攻击者的”反 ss”备用通道

正如 1.3.4 节讲到的，挖矿木马在 hook 掉 `ss`、`netstat`、`lsof` 之后，仍然使用 `/proc/net/tcp` 直连 C2。我们来还原这个过程：

```
# 攻击者手动从 /proc/net/tcp 还原连接列表
awk '$4 == "01" {
  split($2, local, ":")
  split($3, rem, ":")
  # 本地地址解码
  l_ip = local[1]
  l_port = strtonum("0x" local[2])
  # 远端地址解码
  r_ip = rem[1]
  r_port = strtonum("0x" rem[2])
  printf "%s:%d -> %s:%d\n", l_ip, l_port, r_ip, r_port
}' /proc/net/tcp
# 输出: 5D9CXXXX:01BB -> 3E92XXXX:01BB（建立中的可疑连接）
```

这条命令在没有 `ss`、没有 `lsof` 的环境下也能跑——**这就是 `/proc/net/` 在攻击者侧的”原始力”**。

0.3.5.3 用 inode 反查 fd：找到 socket 的”主人”

`/proc/net/tcp` 里的每个连接都有一个 inode。要找到这个 socket 属于哪个进程：

```
# 1. 从 /proc/net/tcp 拿到 inode
inode=$(awk '$4 == "01" && /:01BB$/ {print $10}' /proc/net/tcp)

# 2. 在 /proc/*/fd/* 下找指向这个 inode 的符号链接
for pid in /proc/[0-9]*; do
  ls -la $pid/fd/ 2>/dev/null | grep -E "socket:\[$inode\]"
done
# 输出: lrwx... socket:[1234567]
# 即某个 PID 的 fd 4 是这个 socket

# 3. 确认 PID
grep -l "PPid:" /proc/*/status | head
# 然后看 cmdline 确认进程
```

0.3.5.4 /proc/net/ 里的”非常规文件”

除了常规的 tcp/tcp6/udp/udp6/arp/route/dev，/proc/net/ 下还有几个常被忽略但极有用的文件：

```
# /proc/net/sockstat - 各协议的 socket 计数
cat /proc/net/sockstat
# TCP: inuse 23 orphan 0 tw 0 alloc 28 mem 4
# UDP: inuse 5 ...

# /proc/net/snmp - SNMP 统计
cat /proc/net/snmp | grep -E "Tcp:" -A 30
# 包含所有 TCP 状态机的计数器

# /proc/net/netstat - 详细网络统计
cat /proc/net/netstat | grep -E "Tcp:" -A 30

# /proc/net/rpc - RPC 状态（NFS 经常暴露）

# /proc/net/ip_conntrack（如果启用） - 连接跟踪表
cat /proc/net/ip_conntrack | head
```

这些文件里**常驻的 socket 数量、TCP 重传率、半开连接数**都是非常优秀的可观测性指标——第 6 章会展开。

0.3.5.5 防御侧的”网络交叉验证”

```
# 永远交叉验证
ss -tnp 2>/dev/null        # 用户态视角
cat /proc/net/tcp | wc -l   # 内核视角
# 两者差距太大 → 怀疑 ss 被 hook
```

0.3.6 第五步：隐身形——Rootkit 与 /proc 的对峙

这是攻防最微妙的一步。攻击者已经”住进”机器里，他要做的不是再去攻击别的机器，而是**让自己不被发现**——这就是 rootkit。

0.3.6.1 Rootkit 的两大隐身目标

任何 rootkit 都要解决两个问题：

1. **进程隐身**：让 ps、top 看不到它
2. **网络隐身**：让 ss、netstat 看不到它的连接

解决这两个问题的方式，恰好都与 /proc 对峙。

0.3.6.2 进程隐身 vs /proc/[pid]/

老式 rootkit 用 hook sys_getdents 系统调用的方式，让 getdents（ps 的实现方式）过滤掉自己的 PID。但 /proc/[pid]/ 目录不是 getdents 生成的——它是 procfs 自己的 readdir 实现。

结果：**ls /proc | grep <pid> 总能看到 rootkit 的 PID**——除非攻击者根本**不创建 PID**。

```
# 攻击者的进化路径：
# v1.0 - hook getdents（被 ps 工具绕过）
# v2.0 - hook 整个 procfs 的 readdir（被 /proc/[pid]/maps 暴露）
# v3.0 - 不创建 PID（运行在内核线程，嫁接到现有进程）
# v4.0 - 用 eBPF hook 系统调用（连审计都看不到）
```

v3.0 的 rootkit 正是 1.1 节前言故事里的那个挖矿木马——它通过劫持 kthreadd 的系统调用实现隐身。**但它留下的 fd 痕迹让安全工程师最终抓到了它**。

0.3.6.3 fd 残留：隐身失败的”证据库”

无论 rootkit 多高级，它必须保持至少一个**网络 socket** 与 C2 通信。这个 socket 一定挂在某个用户态进程的 fd 表里——这就是 1.1 节故事里的反转点。

```
# 防御侧的" 金标准" 检查
ls -la /proc/*/fd/ 2>/dev/null | grep socket | sort -u
# 这条命令的" 反隐身" 价值：
# - 即使 ps 被 hook，fd 是进程内核结构里直接管理的
# - 即使 rootkit 不创建 PID，它嫁接的宿主进程 fd 一定会异常
```

0.3.6.4 /proc/kallsyms：内核符号的可见性博弈

/proc/kallsyms 暴露内核的全部符号——函数地址、变量名。这是 rootkit 的”地图”，也是防御者的”导航”。

```
# 攻击者用它做什么
grep "commit_creds\|prepare_kernel_cred" /proc/kallsyms
# 找到 root 提权函数的位置

# 防御者用它做什么
grep -E "^[0-9a-f]+ [tT] " /proc/kallsyms | wc -l
# 数量变化 = 内核模块加载/卸载的迹象
```

内核提供 kptr_restrict 来限制这个文件的读取：

```
# 限制内核符号暴露
echo 2 > /proc/sys/kernel/kptr_restrict
# 0 - 默认，所有人都能读
# 1 - 仅 root（kptr_restrict=1） → %s 地址格式化为 0
# 2 - 永远隐藏
```

0.3.6.5 dmesg 限制：/proc/kmsg 是 kernel 真相窗口

```
# 查看内核日志（默认需要 root 或 CAP_SYSLOG）
cat /proc/kmsg
# 或 dmesg

# 攻击者用它做什么
# 看自己加载的 LKM 是否有 warning
# 看自己的 syscall hook 是否有 stack trace 泄露

# 防御者用它做什么
# 看任何异常事件（kernel oops、segfault、TCP reset 等）
# 检测异常加载的内核模块

echo 1 > /proc/sys/kernel/dmesg_restrict
# 限制普通用户读 dmesg
```

0.3.7 第六步：跨边界——容器与 namespace 越界

0.3.7.1 容器内的 /proc 是”共享的”吗

容器内的 /proc 默认是宿主机 /proc 的一部分——它通过 bind mount 把宿主机 /proc 暴露到容器里。这意味着：

```
# 在容器内执行
ls /proc/1/root/
# 如果 /proc 没有 hidepid，理论上能看到宿主机的根文件系统！
```

容器逃逸的根本机制就是这条链路。1.3.1 节的 runc 三连击只是更精致的变种。

0.3.7.2 /proc/sys/kernel/ns_last_pid 的 namespace 越界

CVE-2022-0185 利用了 /proc/sys/kernel/ns_last_pid 的 namespace 边界错误，让容器内进程能分配 PID 1 的命名空间。修复后这个 CVE 仍然在第 5 章的实战案例里复盘。

```
# 查看当前 namespace
ls -la /proc/$$/ns/
# 输出: lrwxrwxrwx ... cgroup -> cgroup:[4026531835]
#       lrwxrwxrwx ... ipc -> ipc:[4026531839]
#       ...
# 每种 namespace 都有自己的 inode（namespace inode）
```

0.3.7.3 /proc/self/ 的”自指”特性

/proc/self/ 是 /proc/[当前进程 PID]/ 的符号链接。它的特殊之处：

```
# 在容器内，/proc/self 指向容器自己的 PID
# 但如果 /proc 没有正确隔离，/proc/1 可能指向宿主机 PID 1

# 检测自己是否在容器内
cat /proc/1/cgroup
cat /proc/1/sched
# sched 文件包含 cgroup 路径，能精确定位
```

0.3.7.4 K8s/Podman 场景的容器逃逸前置

K8s 集群里，要触发 runc 三连击，攻击者需要：

1. **控制 Pod spec**——能提交带恶意 volumeMount 的 manifest
2. **创建恶意镜像**——entrypoint 必须在容器启动早期执行符号链接替换
3. **保持容器存活**——runc 在容器整个生命周期内持有宿主 procfs 写权限

Podman 的 rootful 模式触发条件相似；rootless 模式（默认用 --userns=keep-id）由于 user namespace 隔离，相对安全。

0.3.7.5 防御侧的”namespace 边界”清单

```
# 1. 容器启动时使用 hidepid
mount -t proc proc /proc -o nosuid,nodev,noexec,relatime,hidepid=2
# hidepid=2 → 其他用户的 PID 不可见
# hidepid=1 → 其他 PID 可见，但 environ/cmdline 不可读

# 2. 容器内禁止访问宿主机 /proc
# 在 K8s 中通过 PodSpec 设置:
#   securityContext:
#     allowPrivilegeEscalation: false
#     capabilities:
#       drop: ["ALL"]

# 3. 检测 namespace 边界
cat /proc/self/status | grep -E "^(NSpid|NStgid|CapEff)"
# NSpid 多个值 = 在多层 namespace 中
```

0.3.8 攻击者视角的总结：六步一气呵成

把六步串起来看，攻击者从拿到最低权限 shell 到完成”环境评估 + 目标定位 + 嗅探密钥 + 内存取证 + C2 隐藏 + 跨边界准备”，**全程依赖 /proc 这一个接口面**。

这给我们的启示是：

> **/proc 既是攻击者的”6 步链”，也是防御者的”6 道关”**——每一道关都是 /proc 上一个具体文件的可观测性。

攻击者步	关键 /proc	防御者对应的可观测信号
摸环境	/proc/version、/proc/cpuinfo、/proc/config.gz	配置变更告警、hidepid=2 隔离
找进程	/proc/[pid]/cmdline、/proc/[pid]/environ	环境变量变化告警、特权进程白名单
读内存	/proc/[pid]/maps、/proc/[pid]/mem	ptrace_scope 限制、/proc/sys/kernel/yama 收紧
看网络	/proc/net/tcp*、/proc/net/arp	连接基线偏离告警、socket inode 异常
隐身形	/proc/kallsyms、/proc/[pid]/fd	异常 fd 告警、kptr_restrict=2
跨边界	/proc/1/root、/proc/sys/kernel/ns_last_pid	容器 mount 配置审计、namespace 边界监控

第 3 章开始，我们会沿着这张”6 道关”地图，搭建 /proc 的五层纵深防御体系。

> **下一章预告**：第 3 章 /proc 五层纵深防御体系——我们会从”减少暴露面”开始，沿着”访问控制 → 行为约束 → 完整性保护 → 可观测性 + 入侵检测”五层，把上一章暴露的攻击者 6 步逐一对应到防御措施。每一层都会给出**可直接落地的配置、命令、代码**。

0.4 第 3 章：/proc 五层纵深防御体系

0.4.1 防御的总原则

第 2 章给出了攻击者 6 步 TTP——但当我们从防御者视角再走一遍，会发现一件耐人寻味的事：

> **攻击者用 /proc 攻，防御者用 /proc 守；攻防双方用的命令甚至很多是同一条。**

这意味着单纯的”增加规则”是防不住的——你限制了 cat /proc/[pid]/environ，攻击者照样能从 /proc/[pid]/status 找到进程身份；你隐藏了所有 PID，他能直接从 /proc/net/tcp 拿网络连接。**/proc 的防御不能是”堵一个洞”，必须是”让攻击者的每一步都付出代价”**。

这就是纵深防御的逻辑。**纵深不是把每层都做厚，而是让攻击者每跨过一层就多暴露一分**。这一章搭建的五层体系，正是按这个原则设计的：

```
第 1 层：减少暴露面 ——— 让攻击者根本看不到 `/proc` 的某些部分
第 2 层：访问控制   ——— 让攻击者即便看到了也读不到
第 3 层：行为约束   ——— 让攻击者即便读到了也不能做危险事
第 4 层：完整性保护 ——— 让攻击者即便做了也留不下隐匿痕迹
第 5 层：可观测性+入侵检测 ——— 让攻击者的一切行动都被记录与告警
```

每层都假设上一层已经失守——这是纵深的核心。

0.4.2 第 1 层：减少暴露面

> **核心思想**：如果攻击者根本看不到 /proc 的某些部分，后面的所有攻击手段都用不上。

0.4.2.1 mount 选项：procfs 暴露面的第一道开关

/proc 的挂载选项直接决定了哪些 /proc 子树对哪些用户可见。mount 命令可以查看当前选项：

```
mount | grep " /proc "
# 典型输出：
# proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

这是个**最弱配置**——proc 默认只带几个选项，但 Linux 支持的 proc 挂载选项远不止这些。让我们一一展开。

0.4.2.2 hidepid：最容易被低估的开关

hidepid 是 procfs 独有的 mount 选项，取值 0/1/2/3：

值	行为
0	默认，所有人都能看到所有 PID
1	其他用户的 PID 不可见（仅自己进程的 PID 可见）
2	其他用户的 PID 目录不可见，但 /proc/[pid] 文件依然可访问（通过 inode 等）
3	完全隐藏（Linux 5.13+）

实战推荐 **hidepid=2**：

```
mount -o remount,hidepid=2 /proc
# 验证
ls -la /proc | head -20
# → 只能看到自己的 PID 和内核线程
```

这步配置能挡掉的攻击场景包括：

- 2.2.2 节”/proc/[pid]/environ 嗅探密钥”——攻击者看不到其他进程的 PID 目录
- 2.2.3 节”高价值进程清单”——攻击者无法枚举其他用户的进程
- 2.3.1 节”/proc/[pid]/maps 进程地图”——无 PID 可查，自然无法读 maps

但它挡不住：

- 攻击者用 **自己的进程**注入到你的服务（比如 web 漏洞反序列化）
- 攻击者通过 **/proc/net/*** 看到的网络连接（hidepid 不影响 /proc/net）
- 攻击者通过 **/proc/sys/** 修改内核参数（如果 /proc/sys 没单独隔离）

0.4.2.3 子集挂载：把 /proc 拆成只读、只写、只读

更激进的方案是**对不同应用挂载不同的 /proc 子集**。这是容器化部署的标准做法。但在动手之前，先把三个 mount 操作的语义理清楚。

mount --bind vs mount --rbind 的差别：

语义	bind	rbind（--bind --recursive）
作用	在新位置创建原路径的镜像入口	在新位置递归创建原路径及其**所有嵌套挂载点**的镜像入口
适用场景	绑一个独立目录（比如 /proc/sys）	绑一个含有其他挂载点的目录树（比如整个 /proc）
/proc 场景的坑	bind /proc 不会带走 /proc/sys、/proc/net 等独立挂载点	rbind /proc 才会递归带所有子挂载

这意味着容器场景里，如果你想“看到宿主机的完整 /proc 子集”，必须用 **mount --rbind /proc /container/proc**；如果只想访问 /proc/sys/kernel，则 mount --bind /proc/sys/kernel /container/... 就够了。

bind 后改 ro 必须分两步——这是实践中一个反复踩到的坑：

```
# ☒ 错误的“一行式”写法：ro 选项会被静默忽略
mount --bind,ro /proc /mnt/proc_ro

# ☒ 正确的两步式写法
mount --bind /proc /mnt/proc_ro
mount -o remount,ro /mnt/proc_ro
# 第一步创建镜像入口，第二步单独 remount 才生效
```

这种两阶段在 /proc 上尤其重要——procfs 上本身有可写文件（/proc/sys/...、/proc/sysrq-trigger 等），如果第一行加了 ro 但内核认为“原文件系统可写”，会**静默接受但不生效**。防御者写脚本时一定要把这个细节写进。

容器场景的标准配置：

```
# 在容器内，把 /proc 拆成多个 mount point
mount -t proc proc /proc -o nosuid,nodev,noexec,relatime,hidepid=2

# 然后再覆盖敏感路径为只读
mount --bind /proc/sys /proc/sys
mount -o remount,ro,bind /proc/sys

mount --bind /proc/sysrq-trigger /proc/sysrq-trigger
mount -o remount,ro,bind /proc/sysrq-trigger

mount --bind /proc/sys/kernel/core_pattern /proc/sys/kernel/core_pattern
mount -o remount,ro,bind /proc/sys/kernel/core_pattern
```

为什么是”覆盖”而不是直接修改？因为 procfs 的某些文件被重新挂载为只读后，内核会拒绝后续的写操作——比如把 /proc/sys/kernel/core_pattern bind 成 ro 之后，runc 三连击的攻击路径就被堵死了。注意：这里的 remount,ro,bind 是**关键组合**——单独的 remount,ro 会作用于原 mount 点，remount,ro,bind 才作用于这个 bind mount 本身。

为什么这个 ro,bind 能在 K8s/Podman 环境里真的挡住 runc 三连击——因为容器进程仍可以读写**容器根文件系统**上的 /proc（容器看到的 /proc/sys/kernel/core_pattern），但这个文件现在是 bind 到宿主的只读挂载点上，任何写尝试都会被内核拒绝（不是返回错误，而是直接 EACCES）。这是一种比 LSM 更原始的“内核拒绝写”。

0.4.2.4 pid namespace：容器场景的终极隔离

pid namespace 是 mount namespace 之外，**/proc 暴露面的另一道阀门**。它决定了容器内 ls /proc 能看到哪些 PID：

```
# 在 K8s PodSpec 里启用 pid namespace
apiVersion: v1
kind: Pod
metadata:
  name: secured-pod
spec:
  securityContext:
    hostPID: false    # 关键！禁止共享宿主 PID namespace
  containers:
  - name: app
    image: myapp
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
```

如果 hostPID: true，**容器内进程就能看到宿主机所有进程**——这是 K8s 里最常见的”看似安全实则裸奔”的配置。

一个容易忽略的点：pid namespace 只控制“能看到哪些 PID”，不控制“能访问哪些 PID 文件”。即使 hostPID: false，如果你在 /proc mount 选项里没有 hidepid=2，攻击者仍可能通过某些被遗露的子目录访问宿主机的 PID。两者必须**同时启用**才是完整防护。

0.4.2.5 屏蔽路径：runc 的 maskedPaths

runc 默认会把以下路径 bind-mount 成 /dev/null（写保护）：

```
# 来自 runc 的 spec.go
/proc/asound
/proc/acpi
/proc/kcore
/proc/keys
/proc/latency_stats
/proc/timer_list
/proc/timer_stats
/proc/sched_debug
/proc/sysrq-trigger
```

```
/proc/mag
/sys/firmware
```

但 1.3.1 节 runc 三连击（CVE-2025-31133/52565/52881）告诉我们——**默认的 maskedPaths 不够**。runc 1.2.8 / 1.3.3 / 1.4.0-rc.3 修复后，**默认列表增加了**：

```
/proc/sys/kernel/core_pattern  # 容器逃逸的关键路径
/proc/sysrq-trigger            # 系统级动作触发
/proc/scsi                     # SCSI 设备信息
```

升级到 patched runc 是最简单的减少暴露面手段。

0.4.2.6 /proc 是在哪里挂上的：启动时序

说了这么多 mount 命令，最后**重要的一点需要看清：这些 mount 在系统启动的哪个阶段发生?**

现代 Linux 启动过程分为 5 个阶段：

```
1. BIOS/UEFI
   ↓ POST + 加载 bootloader
2. GRUB bootloader
   ↓ 加载内核 + initramfs 到内存
3. 内核自解压 + 初始化
   ↓ 【关键】这个阶段内核自己挂载 /proc（类型 = procfs），
   ↓         挂在 initramfs 的临时根上。
   ↓         此时 /proc/sys/、/proc/net/ 这些子挂载都还未创建
4. initramfs 中的 /init 程序
   ↓ 在 tmpfs 上的临时根里做事（加载驱动、加密解密等）
   ↓ 【关键】这里看到的 /proc 是内核挂的裸 procfs，没有 hidepid
   ↓         这里的 mount 命令要么作用于临时根，要么会被后续覆盖
5. pivot_root → systemd 接管
   ↓ 【关键】systemd 重挂 /proc（通常在 proc-fs-namespace.service 或类似路径）
   ↓         这时才会应用 hidepid=、relatime、nodev 等参数
   ↓         这一层是我们加固 /proc 的真正有效点
6. multi-user.target 启动服务
   ↓ 后续所有 mount 命令都在 systemd 的控制下
```

为什么要看清这个时序？因为：

- 在 initramfs 阶段做 /proc 加固是**徒劳的**——systemd 会重新挂载覆盖掉
- hidepid=、默认 maskedPaths 都是 systemd 或它的 unit 文件负责加上的（不是内核启动时默认就有）
- 如果你手工修改 initramfs 里的脚本去设置 hidepid，重启后会被 systemd 覆盖，造成“以为加了实际没加”的隐蔽事故

加固 /proc 的正确路径是：

```
# 1. 修改 systemd 的 /proc 挂载参数
#    文件路径一般是: /etc/systemd/system/proc-fs-namespace.service 或
#    或者在 /etc/fstab 里写一行:
proc /proc proc defaults,nosuid,nodev,noexec,hidepid=2 0 0

# 2. 重启后验证
mount | grep " /proc "
# 应该看到 hidepid=2

# 3. 不要手动 remount（重启会丢）
```

Podman/容器场景例外：容器启动时自己负责挂载 /proc，不走 systemd。但业务容器（在 Podman rootful 模式下）的 /proc 是 runc 挂的，跟宿主 systemd 无关。所以容器里做的 mount 不会反向影响宿主。

0.4.2.7 实战：审计你机器上的 /proc 暴露面

```
#!/bin/bash
# /usr/local/bin/proc_exposure_audit.sh
```

```
# 审计当前机器 /proc 的暴露面配置

echo "=== 当前 /proc 挂载配置 ==="
mount | grep " /proc "
echo ""

echo "=== hidepid 是否启用 ==="
mount | grep " /proc " | grep -oE "hidepid=[0-9]" || echo "☒ 未启用 hidepid"
echo ""

echo "=== 检查 hostPID（容器场景） ==="
if [ -f /.dockerenv ]; then
  echo " 在 Docker 容器内，检查 hostPID"
  cat /proc/1/status | grep -E "^(NSpid|Pid)" | head -5
fi
echo ""

echo "=== 危险文件是否可写 ==="
for f in /proc/sys/kernel/core_pattern /proc/sysrq-trigger /proc/sys/kernel/modprobe; do
  if [ -w "$f" ]; then
    echo "⚠  $f 可写"
  else
    echo "☒ $f 只读"
  fi
done
echo ""

echo "=== 普通用户能看到的进程数 ==="
su - nobody -s /bin/bash -c "ls /proc/[0-9]* 2>/dev/null | wc -l"
echo ""

echo "=== /proc/net 是否对普通用户可读 ==="
su - nobody -s /bin/bash -c "cat /proc/net/tcp | wc -l"
```

把这个脚本放进日常巡检，**任何”hidepid 没启用”或”core_pattern 可写”的机器都应该立刻处理**。

0.4.2.8 减少暴露面的”代价”

需要清醒看到：**/proc 的暴露面减少是有副作用的**。

- hidepid=2 会让 /usr/bin/pkill 等工具失效（因为看不到其他 PID）
- 子集挂载会让 /proc/[pid]/comm 等只读信息也无法访问（影响一些监控工具）
- 屏蔽 maskedPaths 会让 cat /proc/kmsg 等调试工具失效

我们推荐”**逐步收紧**”的策略：

```
# 第 1 周：只启用 hidepid=1
mount -o remount,hidepid=1 /proc
# 等所有监控告警系统适配

# 第 2 周：升级到 hidepid=2
mount -o remount,hidepid=2 /proc

# 第 3 周：加固 maskedPaths（升级 runc 或手工 mount bind）

# 第 4 周：增加 read-only bind mount（针对 core_pattern 等）
```

这个节奏比”一次到位”更安全——**每一步都有回滚余地**。

0.4.3 第 2 层：访问控制

核心思想：攻击者即便看到了 /proc 的某些部分，也读不到关键内容。

0.4.3.1 /proc/[pid]/* 的文件权限矩阵

/proc/[pid]/ 下每个文件的权限都不同。理解这张矩阵是访问控制的基础：

```
# 用 stat 看每个文件的权限
for f in cmdline environ status maps fd exe cwd; do
  pid=$$
  ls -la /proc/$pid/$f 2>/dev/null
done
# 典型输出：
# -r-------- 1 axu axu 0 ... cmdline       # 仅本人可读
# -r-------- 1 axu axu 0 ... environ       # 仅本人可读
# -r--r--r-- 1 axu axu 0 ... status        # 所有人可读
# -r--r--r-- 1 axu axu 0 ... maps          # 所有人可读（受 ptrace 限制）
# dr-x------ 2 axu axu 0 ... fd/           # 仅本人可读
# lrwxrwxrwx 1 axu axu 0 ... exe           # 所有人可读
# lrwxrwxrwx 1 axu axu 0 ... cwd           # 所有人可读
```

注意几个关键点：

- **environ** 默认仅本人可读——但 root 启动的进程例外（root 的 environ 所有人可读）
- **status** 所有人可读——包括 Pid、PPid、Uid、Gid 等身份信息
- **maps** 所有人可读——但要 mem 读写还得过 ptrace 这关
- **fd/** 默认仅本人可读——这正是 rootkit 残留 fd 检查的难点（普通用户查不到别人的 fd）

0.4.3.2 capability 拆分：从 root 到 41 个细分权限

传统 Unix 把权限分为 root vs 普通用户。Linux 把 root 拆成 **41 个 capability**：

```
# 查看当前进程的 capability 集合
cat /proc/self/status | grep ^Cap
# CapInh: 0000000000000000
# CapPrm: 0000000000000000
# CapEff: 0000000000000000
# CapBnd: 000001ffffffffff
# CapAmb: 0000000000000000
```

跟 /proc 直接相关的几个 capability：

Capability	作用
CAP_SYS_PTRACE	允许 ptrace 任意进程
CAP_SYS_ADMIN	一揽子权限（包括 mount、setns、bpf 等）
CAP_DAC_READ_SEARCH	绕过文件读权限检查
CAP_SYS_RESOURCE	覆盖资源限制
CAP_NET_ADMIN	网络管理（创建 tun 设备——CVE-2025-40271 的触发条件）

容器化部署的标准做法是**彻底 drop 全部 capability，只 add 必需的**：

```
# K8s PodSpec
securityContext:
  capabilities:
    drop: ["ALL"]
    add: ["NET_BIND_SERVICE"]  # 仅 web 服务需要的
```

但这还不够——**CAP_SYS_PTRACE 在很多容器镜像里默认没被 drop**，导致 2.3.3 节的”内存注入”仍然可行。

0.4.3.3 AppArmor / SELinux：基于 LSM 的 /proc 访问控制

AppArmor profile 示例（限制 nginx 进程对 /proc 的访问）：

```
# /etc/apparmor.d/local/nginx
#include <tunables/global>
```

```
profile nginx-restricted flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  #include <abstractions/nameservice>

  # 允许读 /proc/self/status（监控用）
  /proc/self/status r,
  # 允许读 /proc/loadavg
  /proc/loadavg r,
  /proc/meminfo r,
  /proc/stat r,

  # 关键：禁止读其他进程的 /proc/[pid]/*
  deny /proc/[0-9]*/environ r,
  deny /proc/[0-9]*/cmdline r,
  deny /proc/[0-9]*/maps r,
  deny /proc/[0-9]*/mem rw,
  deny /proc/[0-9]*/fd/* r,

  # 关键：禁止写 /proc/sys/*
  deny /proc/sys/** w,
  deny /proc/sysrq-trigger w,
  deny /proc/sys/kernel/core_pattern w,
}
```

SELinux 规则示例（同样限制）：

```
# 禁止 httpd_t 域写 proc sysctl
allow httpd_t proc_t : file { read getattr };
deny  httpd_t sysctl_t : file { write create setattr };
```

0.4.3.4 seccomp：阻断读 /proc/[pid]/mem

seccomp（secure computing mode）是内核的 syscall 过滤器。它对 /proc 的保护特别直接——**open(/proc/[pid]/mem) 这个 syscall 可以直接被 seccomp 拦截**：

```
// seccomp 规则示例（用 libseccomp）
#include <seccomp.h>

scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);

// 禁止 openat 调用打开其他进程的 mem
seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EACCES),
    SCMP_SYS(openat), 1,
    SCMP_CSTR(2));  // 第二个参数：路径

// 禁止 ptrace
seccomp_rule_add(ctx, SCMP_ACT_KILL_PROCESS,
    SCMP_SYS(ptrace), 0);

// 禁止 process_vm_readv（跨进程内存读）
seccomp_rule_add(ctx, SCMP_ACT_KILL_PROCESS,
    SCMP_SYS(process_vm_readv), 0);

// 加载
seccomp_load(ctx);
```

这条规则直接对应 2.3.2 节”/proc/[pid]/mem 进程内存取证”——即便攻击者有读 /proc/[pid]/maps 的权限，也读不了 mem。

0.4.3.5 kernel.yama.ptrace_scope：从内核层限制 ptrace

ptrace_scope 是 Linux 内核的 ptrace 限制开关：

```
# 查看当前值
cat /proc/sys/kernel/yama/ptrace_scope
# 默认是 0（宽松）→ 1（限制）→ 2（仅 root）→ 3（完全禁用）

# 推荐：1 或 2
echo 1 > /proc/sys/kernel/yama/ptrace_scope
```

这条配置挡住的攻击场景：

- 2.3.2 节”gcore 取进程 core dump”
- 2.3.3 节”gdb -p 写进程内存”
- CVE-2026-46333 的”race window”利用（虽然不能完全防住 race，但能缩窄窗口）

0.4.3.6 /proc/sys/kernel/ 关键权限设置

/proc/sys/kernel/ 下有几个文件直接关系到 /proc 暴露面：

```
# 1. 隐藏内核符号（攻击者用它定位提权函数）
echo 2 > /proc/sys/kernel/kptr_restrict
# 0 - 所有人都能读（默认）
# 1 - root 才能读
# 2 - 永远隐藏

# 2. 限制 dmesg（隐藏内核日志）
echo 1 > /proc/sys/kernel/dmesg_restrict
# 0 - 所有人都能读
# 1 - 仅 root / CAP_SYSLOG

# 3. 限制 usermodehelper（防止内核调用任意用户态程序）
echo 1 > /proc/sys/kernel/usermodehelper/
# 这个值控制 modprobe 等行为，是 CVE-2025-31133 的一类缓解

# 4. BPF 限制（防止非特权用户用 BPF）
echo 1 > /proc/sys/kernel/unprivileged_bpf_disabled
# 0 - 允许（默认）
# 1 - 仅特权用户
# 2 - 永远禁止

# 5. kexec 限制
echo 1 > /proc/sys/kernel/kexec_load_disabled

# 6. 限制 SysRq（防止攻击者触发系统级动作）
echo 0 > /proc/sys/kernel/sysrq
# 0 - 完全禁用 SysRq（对应 1.3.1 节的 sysrq-trigger 路径）
```

这些配置是 procfs 的”自我防护”——它们不是限制 /proc 本身，而是限制 /proc 暴露出的”危险接口”。

0.4.3.7 访问控制层的”代价”

访问控制不是”越严越好”。每个限制都有副作用：

- ptrace_scope=1 会让 strace/ltrace/gdb 等调试工具失效
- seccomp 太严会让某些合法 syscall 失败
- AppArmor profile 一旦写错，进程可能无法启动

我们推荐”**白名单模式**”：先 deny 全部，再 add 必需的。

```
# 错误的做法：deny 黑名单
deny /proc/[0-9]*/environ r,
# 问题：攻击者用其他文件绕过（maps + status + stat）

# 正确的做法：deny 全部，allow 白名单
deny /proc/[0-9]*/** rw,
```

```
allow /proc/self/status r,
allow /proc/loadavg r,
```

0.4.3.8 实战：/proc 访问控制的”一键加固脚本”

```
#!/bin/bash
# /usr/local/bin/proc_access_hardening.sh
# /proc 访问控制一键加固

set -euo pipefail

echo "=== 启用 hidepid ==="
mount -o remount,hidepid=2 /proc

echo "=== 设置 ptrace_scope ==="
echo 1 > /proc/sys/kernel/yama/ptrace_scope

echo "=== 隐藏内核符号 ==="
echo 2 > /proc/sys/kernel/kptr_restrict

echo "=== 限制 dmesg ==="
echo 1 > /proc/sys/kernel/dmesg_restrict

echo "=== 限制 usermodehelper ==="
echo 1 > /proc/sys/kernel/usermodehelper/bset
echo 1 > /proc/sys/kernel/usermodehelper/inheritable

echo "=== 禁用 BPF（普通用户） ==="
echo 1 > /proc/sys/kernel/unprivileged_bpf_disabled

echo "=== 禁用 SysRq ==="
echo 0 > /proc/sys/kernel/sysrq

echo "=== 验证 ==="
mount | grep " /proc "
echo ""

echo " 加固完成。建议："
echo "1. 重启前先保持另一个 SSH 会话，验证仍能登录"
echo "2. 业务进程如果使用 strace/gdb，需要单独适配"
echo "3. 监控告警系统可能需要重新适配（hidepid 隐藏了 PID）"
```

这个脚本可以直接放进 Ansible/Chef 的初始化 playbook，作为新机器的”开箱即用”配置。

0.4.4 第 3 层：行为约束

核心思想：在攻击者“能访问 /proc ”的前提下，限制他能做的具体动作。

行为约束与访问控制的区别是粒度：访问控制决定“能不能读”，行为约束决定“读完之后能不能接 socket / 不能 fork / 不能动某些文件”。**它是 AppArmor + seccomp + landlock 这套组合拳的主战场**。

0.4.4.1 AppArmor/SELinux 完整 profile 模板

3.2.3 节给过 nginx 的 mini profile。这一节给一个更完整的 **业务进程参考模板**，适用于任何“读 /proc/self、但禁止读其他进程”的业务应用。

```
# /etc/apparmor.d/local/myapp-restricted
#include <tunables/global>

profile myapp-restricted flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
```

```
  #include <abstractions/nameservice>

  # 自身 /proc 只读
  /proc/self/status r,
  /proc/self/comm r,
  /proc/self/loginuid r,
  /proc/self/oom_score r,
  /proc/self/cgroup r,

  # 全局只读信号
  /proc/loadavg r,
  /proc/meminfo r,
  /proc/stat r,
  /proc/uptime r,
  /proc/version r,
  /proc/filesystems r,
  /proc/sys/kernel/random/uuid r,    # libuuid 使用

  # 关键：禁止读其他进程的 /proc/[pid]/*
  deny /proc/[0-9]*/** rw,
  deny /proc/[0-9]*/environ r,
  deny /proc/[0-9]*/cmdline r,
  deny /proc/[0-9]*/maps r,
  deny /proc/[0-9]*/mem rw,
  deny /proc/[0-9]*/fd/ r,

  # 禁止写 /proc/sys/**（对应 3.1.5 的防御目标）
  deny /proc/sys/** w,
  deny /proc/sysrq-trigger w,
  deny /proc/sys/kernel/core_pattern w,
  deny /proc/sys/kernel/modprobe w,
  deny /proc/sys/kernel/hotplug w,

  # 禁止 mount/umount（rootkit 常用）
  deny capability sys_admin,

  # 禁止 ptrace 与 process_vm_*（对应 2.3 节内存注入）
  deny capability sys_ptrace,
  deny ptrace,
  deny ptrace_peer,
  deny signal,

  # 业务需要的能力（仅举几个例）
  capability net_bind_service,
  capability dac_read_search,    # 如果需要读 protected config
  capability sys_resource,      # 调整 rlimit
}
```

SELinux 等价规则（以 httpd_t 为例）：

```
# 禁止 httpd 读其他进程的 /proc
allow httpd_t proc_t:dir { getattr search };
deny  httpd_t proc_t:file { read };

# 禁止写 /proc/sys
deny  httpd_t sysctl_t:file { write create setattr };

# 禁止 ptrace
deny  httpu_t process:ptrace;
```

0.4.4.2 seccomp 与 BPF LSM 的组合

3.2.4 节给了一个 libseccomp 的 mini 示例。这一节说**为什么 seccomp + BPF LSM 才是当代完整防御**。

seccomp 的边界：它只能拦截 syscall，不能拦截“打开某个路径”。对于“能不能 open(/proc/[pid]/mem)”这种问题，seccomp 只能“拒绝所有 openat 调用”（太粗）或者“依赖 path name 检测”（路径名是 syscall 参数，seccomp BPF 能读）。

BPF LSM 是补充：内核从 5.7 开始支持 BPF LSM，允许你**在文件操作处拦截**（不是 syscall 层）。这意味着你可以写出：

```
// BPF LSM 示例：拒绝打开非自身进程的 mem
SEC("lsm/file_open")
int BPF_PROG(restrict_mem_open, struct file *file) {
    struct task_struct *task = bpf_get_current_task();
    // 获取当前进程的 PID
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    // 从 file 获取 inode，检查是否在 /proc/[pid]/mem
    // 如果不是当前 PID 的 mem → 拒绝
    // ...
    return -EPERM;
}
```

实战组合：- **seccomp**：负责 syscall 级别的粗粒度拦截（禁用 ptrace、process_vm_*、kexec_load 等）- **BPF LSM**：负责文件级别的细粒度拦截（基于路径、inode、进程上下文）

这是现代 Falco/Tetragon 这类可观测性平台的技术内核。

0.4.4.3 容器场景的 RuntimeDefault seccomp profile

containerd 默认装载一个 RuntimeDefault seccomp profile，包含 **约 50+ 个 syscall 的默认拒绝规则**。该 profile 文件位于 containerd 源码里：

https://github.com/containerd/containerd/blob/main/contrib/seccomp/seccomp_default.go

核心拦截包括：

```
// 简化的 default profile 逻辑
{
    Name:   "ptrace",
    Action: SECCOMP_ACT_ERRNO,  // 拒绝所有 ptrace
},
{
    Name:   "process_vm_readv",
    Action: SECCOMP_ACT_ERRNO,  // 拒绝跨进程内存读
},
{
    Name:   "kexec_load",
    Action: SECCOMP_ACT_ERRNO,
},
{
    Name:   "mount",
    Action: SECCOMP_ACT_ERRNO,  // 容器不能 mount（除非明确允许）
},
{
    Name:   "umount2",
    Action: SECCOMP_ACT_ERRNO,
},
{
    Name:   "clone",
    Args: []argument{
        {
            Index: 0,
            Op:    SECCOMP_ARG_MASK,
            UMask: ^uint64(CLONE_NEWNS|CLONE_NEWUSER|CLONE_NEWPID),
            // 仅允许标准 fork，不允许创建新 namespace
        },
    },
```

```
    Action: SECCOMP_ACT_ALLOW,
},
```

推荐：所有 K8s Pod 默认装载 `RuntimeDefault` profile，自定义需求用 `securityContext.seccompProfile.type: Localhost` 加载专门写的 profile。

```
# K8s PodSpec
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

0.4.4.4 landlock：用户态 LSM 的新边界

landlock 是 Linux 5.13+ 引入的**用户态 LSM**——业务进程可以在运行时动态限制自己的文件系统访问。

```
// landlock 示例：限制进程只能读 /proc/self
#include <linux/landlock.h>

struct landlock_path_beneath_attr attr = {
    .allowed_access = LANDLOCK_ACCESS_FS_READ_FILE,
    .parent_fd = open("/proc/self", O_PATH),
};

int ruleset_fd = landlock_create_ruleset(NULL, 0, LANDLOCK_CTL_FLAGS_NONE);
landlock_add_rule(ruleset_fd, LANDLOCK_KEY_PATH_BENEATH, &attr, sizeof(attr));
landlock_restrict_self(ruleset_fd, LANDLOCK_CTL_FLAGS_NONE);
```

landlock 与 seccomp 的区别：- seccomp：静态，进程启动时加载 - landlock：**动态**，进程运行时可以“套上笼子” - landlock 只限制文件系统访问（不能限制 syscall、不能限制网络）- landlock 需要 Linux 5.13+，部分嵌入式内核未启用

实战场景：一个 web 渲染服务可以**启动后**用 landlock 限制自己只能读 `/proc/self`、不能读其他进程——即使服务被 RCE 拿下，攻击者也无法读到其他进程的密钥。

0.4.4.5 静态防御 vs 动态防御的边界

静态防御（AppArmor/SELinux/seccomp/landlock）的特点是 **“白名单模式”** + **“预先定义”**：- 优点：性能开销几乎为零；规则一旦写好就稳定 - 缺点：未知场景 / 0day 不会被拦住

动态防御（Falco/Tetragon/CrowdSec）的特点是 **“异常检测”** + **“运行时决策”**：- 优点：能识别未知攻击 - 缺点：依赖基线、需要持续训练、有误报

两者必须结合——静态防御挡 99% 的已知攻击，动态防御发现剩余的 1% 异常。第 5 章的入侵检测系统会展开动态防御。

0.4.4.6 行为约束的”代价”

- AppArmor profile 一旦写错会**直接拒绝进程启动**——务必加 `audit` mode 测试一段时间
- seccomp profile 错误会让进程**部分功能失效**——务必先在 audit mode（`SECCOMP_ACT_LOG`）跑 1-2 周
- landlock 动态限制会被子进程继承——需要规划好子进程的生命周期
- BPF LSM 需要内核编译选项 `CONFIG_BPF_LSM=y`——并非所有发行版默认启用

推荐上线路径：

```
# 第 1 周：AppArmor enforce 模式（以“违规即拒绝”为主）
aa-enforce /etc/apparmor.d/local/myapp-restricted

# 第 2 周：启用 RuntimeDefault seccomp（K8s 集群级）
kubectl label namespace my-namespace pod-security.kubernetes.io/enforce=restricted

# 第 3 周：上 BPF LSM（需要内核支持）
bpftool prog loadall /usr/share/bpf/restrict-mem.bpf.o /sys/fs/bpf/

# 第 4 周：上 landlock（仅对个别高安全要求服务）
```

业务方要警惕：这一层的“代价”不是成本，而是**错误配置的恢复成本**。一个被错误 enforce 的 AppArmor profile 可能让全站 502；一个错误的 seccomp profile 可能让 Node 服务起不来。务必先 audit，后 enforce。

0.4.5 第 4 层：完整性保护

核心思想：让攻击者即便做了某些修改动作，也留下不可抹除的证据。

前三层都是“限制能不能”，这一层是**“留下做了什么”**。它的逻辑是：**/proc 里的某些信息一旦被恶意篡改（比如隐藏 PID、隐藏模块），必须能被检测出来**。

0.4.5.1 /proc/kallsyms 隐藏：攻防双视角

在 2.5.4 节说过，/proc/kallsyms 是 rootkit 的”导航地图”。防御手段是把符号隐藏：

```
# 默认隐藏所有符号
echo 2 > /proc/sys/kernel/kptr_restrict
# 0 - 默认，所有人都能读
# 1 - 仅 root 能读
# 2 - 永远隐藏（%s 格式化为 0）

# 验证
cat /proc/kallsyms | head -5
# 0x0000000000000000 T startup_64
# 0x0000000000000000 T secondary_startup_64
# 所有地址都是 0 → 攻击者拿不到函数地址
```

但**攻防双视角的细节**在这里出现：

- **防御侧**：kptr_restrict=2 隐藏符号，让攻击者**难以定位内核函数地址**
- **攻击侧**：攻击者可以**自己 root 后**强制把这个值改回去

```
# 如果攻击者拿到 root，可以一键恢复符号可见
echo 0 > /proc/sys/kernel/kptr_restrict
# 然后继续读 /proc/kallsyms
```

所以 kptr_restrict 不是终极防御，只是**“让普通用户拿不到符号”**。**真正的纵深防御是：让 root 也不能轻易改 sysctl**——这要靠 /proc/sys 的读写权限。

```
# 限制 /proc/sys 的写权限（root 也需要过能力检查）
# /etc/fstab 加一行
proc /proc/sys proc nosuid,nodev,noexec,remount,hidepid=2 0 0
# 或者用 mount propagation（3.5.4 节详述）
```

0.4.5.2 /proc/modules：内核模块的可观测性

内核模块加载/卸载后，/proc/modules 立即反映：

```
# 查看当前加载的模块
cat /proc/modules
# 输出格式：模块名 大小 引用计数 依赖列表 [状态] 地址

# 可疑模块检测（手工方法）
awk '{print $1}' /proc/modules | while read mod; do
  # 检查模块是否在合法列表中
  if ! grep -q "^$mod$" /etc/allowed-kernel-modules.txt; then
    echo "⚠  可疑模块：$mod"
  fi
done
```

更严谨的做法是 **Linux 内核模块签名（CONFIG_MODULE_SIG）**：

```
# 启用模块签名验证
echo 1 > /proc/sys/kernel/modules_disabled
# 启用后，只有签名过的模块才能被加载
```

```
# 验证当前状态
cat /proc/sys/kernel/modules_disabled
# 1 = 完全禁用模块加载
# -1 = 不可逆禁用
```

但**强烈建议不要在生产启用 modules_disabled=1**——这个开关一旦设置，**重启前无法关闭**。一旦误启用，整个系统升级、驱动调试都会失能。

0.4.5.3 IMA（Integrity Measurement Architecture）：对 /proc 关键路径度量

IMA 是 Linux 内核的完整性度量框架。它可以在**关键文件被访问时**计算哈希并记录：

```
# 启用 IMA 度量
echo "ima" > /sys/kernel/security/lsm

# 配置度量规则（/etc/ima/ima-policy）
measure func=BPRM_CHECK mask=MAY_EXEC
measure func=FILE_CHECK mask=MAY_READ
# ↑ 度量所有被执行的二进制和被读取的关键文件

# 查看度量结果
cat /sys/kernel/security/integrity/ima/ascii_runtime_measurements
# 输出示例：
# 10 ... ima-ng sha256:abc... /usr/sbin/sshd
# 10 ... ima-ng sha256:def... /etc/ssh/sshd_config
```

对 /proc 的意义：如果攻击者把 /usr/sbin/sshd 替换为 rootkit 版本，IMA 会记录**“这次执行的不是合法 sshd”**——即便攻击者清理了 /proc，IMA 度量已写入 TPM/远程日志。

0.4.5.4 /proc/[pid]/maps 的不可信化

攻击者通过篡改 task_struct->mm->mmap 可以伪造 maps。但内核提供了一些手段让“伪造”变得困难：

```
# 检查进程 maps 是否包含“看起来不像”的内存段
cat /proc/$$/maps | awk '$5 == "---p" {print}' | head
# ---p 00000000 00:00 0
# 这种全零地址 + anonymous private 是常见的注入点
```

主动防御：

```
# 1. 启用 ASLR（默认开了）
cat /proc/sys/kernel/randomize_va_space
# 2 = 完整 ASLR（推荐）

# 2. 限制非特权用户执行无主内存页（防止堆栈注入）
echo 2 > /proc/sys/vm/mmap_min_addr
# 0 = 允许映射任何地址（危险）
# 65536 = 推荐起点（64KB）
```

0.4.5.5 /proc/net/tcp 的不可信化

同样,攻击者可以 hook tcp_seq_show 让 /proc/net/tcp 显示“被过滤过的”连接。防御侧使用 **eBPF + immutable file**：

```
# 把 /proc/net/tcp 设为 immutable（chattr +i）
chattr +i /proc/net/tcp
# ☒ 这条命令不成立——procfs 文件不能用 chattr

# 正确的做法：用 eBPF 持续监控 /proc/net/tcp 的访问
# 检测“是否有进程读 /proc/net/tcp 但实际并没有真实的 netstat 调用”
```

实战中，**/proc/net 的篡改难以做完整**——因为 netstat/ss/lsof 都从 /proc/net/tcp 读数据，如果改坏了，所有这些工具都会异常。所以攻击者通常选择**“添加 filter”而不是“替换内容”**——这会在第 5.5.6 节的异常画像里被识别。

0.4.5.6 dm-verity + read-only rootfs：从源头防篡改

如果 /proc 暴露的信息本身就被恶意篡改（比如 /usr/bin/ps 被换成过滤版本），那所有 /proc 的可观测性都失效。这时候需要**从源头防篡改**：

```
# 1. dm-verity 给 rootfs 提供块级完整性
#    /etc/dm-verity.conf 配置每个块的哈希
#    内核启动时验证所有块是否匹配

# 2. read-only rootfs
mount -o remount,ro /
# 整个 rootfs 不可写，攻击者替换任何文件都会失败

# 3. 只读挂载 /proc
mount -o remount,ro,bind /proc/sys
# 节详述过这个二阶段 mount
```

0.4.5.7 完整性保护的“代价”

完整性保护是最“重型”的防御手段。它的代价包括：

- **IMA 度量带来 5-15% 的性能开销**（依赖度量频率）
- **dm-verity 重启后无法在线修改 rootfs**——所有配置变更需要走 A/B 分区升级
- **modules_disabled=1 是不可逆的**（重启前不能关闭）
- **/proc 某些只读 bind mount 会让某些调试工具失效**（比如不能清空 /proc/sys/vm/drop_caches）

推荐上线顺序：

```
# 第 1 个月：启用 ASLR（默认在）+ mmap_min_addr
echo 2 > /proc/sys/kernel/randomize_va_space
echo 65536 > /proc/sys/vm/mmap_min_addr

# 第 2 个月：启用 IMA 度量 + 验证
# （在测试环境验证 30 天后再上生产）

# 第 3 个月：dm-verity（仅新机器适用，老机器需要重新制作镜像）
```

业务方需要知道的：完整性保护是“最后一道关”——前面的 1-3 层都失败后，攻击者才会触及这一层。在这一步上付出性能成本是值得的。

0.4.6 第 5 层：可观测性 + 入侵检测

> **核心思想**：让攻击者的一切行动都留下记录，被我们看见。

这一层是“**看**”的体系。前四层是“防”，这一层是“看见”。两者结合才是完整纵深——**因为没有防得住的攻击，只有没看见的攻击**。

0.4.6.1 auditd 规则集：把 /proc 关键事件全部记录

auditd 是 Linux 的审计子系统。它可以在**/proc 路径被访问时**记录事件：

```
# /etc/audit/rules.d/proc.rules
# 监控所有 /proc/[pid]/* 的读访问
-w /proc/[0-9]*/ -p r -k proc_read

# 监控 /proc/sys/ 写入
-w /proc/sys/ -p wa -k proc_sys_write

# 监控敏感 sysctl
-w /proc/sys/kernel/core_pattern -p wa -k core_pattern
-w /proc/sysrq-trigger -p wa -k sysrq
-w /proc/sys/kernel/modprobe -p wa -k modprobe
```

```
# 监控隐藏进程的可能路径
-w /proc/[0-9]*/cmdline -p r -k cmdline_read
-w /proc/[0-9]*/environ -p r -k environ_read
-w /proc/[0-9]*/maps -p r -k maps_read

# 监控系统调用
-a always,exit -F arch=b64 -S ptrace -k ptrace
-a always,exit -F arch=b64 -S process_vm_readv -k proc_mem_read
-a always,exit -F arch=b64 -S init_module -k module_load
-a always,exit -F arch=b64 -S delete_module -k module_unload

# 应用规则
auditctl -R /etc/audit/rules.d/proc.rules
```

ausearch 查询:

```
# 查所有读 /proc/[pid]/environ 的事件
ausearch -k environ_read --start recent

# 查所有写入 core_pattern 的事件
ausearch -k core_pattern --start recent

# 查所有 ptrace 事件
ausearch -k ptrace --start recent

# 查所有加载内核模块的事件
ausearch -k module_load --start recent
```

auditd 的局限：性能开销大。上面这套规则会让 ps、top 这类每秒钟读 /proc 几十次的工具产生**海量日志**。建议：

```
# 在生产用更精细的过滤
# 只记录 uid>=1000 的进程读其他进程的 /proc
-a always,exit -F arch=b64 -S openat -F path=/proc/[0-9]*/ -F auid>=1000 -k proc_read_user
```

0.4.6.2 eBPF：动态追踪 /proc 访问

eBPF 比 auditd 更轻量，可以在 syscall 层做实时追踪：

```
// /proc/self/comm 的 eBPF trace
SEC("tracepoint/syscalls/sys_enter_openat")
int trace_openat(struct trace_event_raw_sys_enter *ctx) {
    char filename[256];
    bpf_probe_read_user_str(filename, sizeof(filename), (char *)ctx->args[1]);

    // 仅追踪读 /proc/[pid]/* 的事件
    if (strncmp(filename, "/proc/", 6) != 0) return 0;
    if (strncmp(filename + 6, "self", 4) == 0) return 0;  // 跳过自身

    // 输出到 perf event
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("PID %d opened %s\\n", pid, filename);
    return 0;
}
```

用 bpftool 加载并查看：

```
# 编译
clang -O2 -target bpf -c proc_trace.bpf.c -o proc_trace.bpf.o

# 加载
bpftool prog load proc_trace.bpf.o /sys/fs/bpf/proc_trace
bpftool prog attach /sys/fs/bpf/proc_trace tracepoint/syscalls/sys_enter_openat

# 查看追踪输出
cat /sys/kernel/debug/tracing/trace_pipe | grep "PID"
```

eBPF 的优势：性能开销比 auditd 低一个数量级。**eBPF 的劣势**：需要内核版本 4.4+，并且 BPF 程序一旦写错会内核 panic（所以要在测试环境充分验证）。

0.4.6.3 Falco / Tetragon：/proc 视角的入侵检测

Falco 是 CNCF 的开源入侵检测工具。它默认的规则集里就有大量 /proc 相关检测：

```
# Falco 默认规则：检测容器内写 core_pattern
- rule: Container Drift Detected (Modify_core_pattern)
  desc: Detect modification of core_pattern in container
  condition: >
    container and
    fd.name=/proc/sys/kernel/core_pattern and
    evt.type in (write, open, openat) and
    evt.is_open_write=true
  output: >
    Core pattern modified in container
    (user=%user.name command=%proc.cmdline pid=%proc.pid)
  priority: CRITICAL

# Falco 默认规则：检测敏感文件被读
- rule: Read sensitive file trusted after startup
  desc: >
    An attempt to read /proc/self/environ from a non-tty process
  condition: >
    proc.name != sshd and fd.name=/proc/self/environ
  output: >
    Sensitive file read (user=%user.name command=%proc.cmdline)
  priority: WARNING
```

Tetragon（Cilium 出品）是更现代的 eBPF-based 工具，可以在 K8s 集群级别启用：

```
# Tetragon TracingPolicy：监控所有读 /proc/[pid]/mem 的尝试
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: proc-mem-read-monitor
spec:
  kprobes:
  - call: "sys_openat"
    syscall: true
    args:
    - index: 1
      type: "const_buf"
    selectors:
    - matchArgs:
      - index: 1
        operator: "Contains"
        values:
        - "/proc/"
        - "/mem"
      matchActions:
      - action: Sigkill
        # 真正可疑就杀进程
      - action: Post
        # 同时打日志
```

0.4.6.4 mount propagation：容器间隔离的隐蔽细节

这一节补一个很多人在生产中踩过的坑：**mount propagation**。

容器内做的 mount 默认是 `private`（不传播）。但有时候你会希望容器间共享 mount（比如 `systemd-resolved` 共享 `/run/systemd/resolve`）——这时会启用 `shared` 或 `slave` propagation。

runc 三连击（CVE-2025-52881）的根因之一就是 mount propagation 配置错误——攻击者通过共享挂载点把宿主的 /proc 写入重定向到自己的路径上。

```
# 查看当前 namespace 的 propagation 类型
findmnt -o TARGET,PROPAGATION
# 输出:
# /   shared
# /proc private    # ← 容器内独立
# /sys  private

# 验证 CVE-2025-52881 缓解: 宿主 /proc 必须是 private
mount | grep " /proc " | grep -v "private"
# 如果有任何非 private 的输出 → 风险仍存在
```

修复方式:

```
# 在容器启动脚本里（runc 的 spec）
"mounts": [
  {
    "destination": "/proc",
    "type": "proc",
    "source": "proc",
    "options": ["nosuid", "nodev", "noexec", "relatime", "hidepid=2"]
  }
]
# propagation 字段默认是 private，runc 1.2.8+ 修复后会强制约束
```

0.4.6.5 /proc 视角的异常画像（基线 + 偏离告警）

这一层是**“把 /proc 变成可观测性指标”**的核心:

```
# Prometheus 查询样例

# 1. 异常 fd 数量（攻击者可能在进程内藏 socket）
node_processes_open_fds{job="prod"} > 1000

# 2. 异常网络连接（监控 /proc/net/tcp 的 ESTABLISHED 计数）
node_netstat_Tcp_CurrEstab > 10000

# 3. 进程数异常增加（可能是 rootkit fork）
node_processes_state{state="Zombie"} > 100

# 4. /proc/[pid]/comm 为空的进程（隐藏的进程）
count(
  count_over_time(node_processes_state[5m]) by (pid)
) - count(
  count_over_time(node_processes_name[5m]) by (pid)
)
```

关键指标（建议接入 Prometheus）:

指标	来源	告警条件
进程总数	`ls /proc/[0-9]* | wc -l`	同比偏离 $> 50\%$
进程创建速率	/proc/loadavg	持续 $> 100/s$
TCP 连接数	/proc/net/tcp	ESTABLISHED > 5000
/proc/[pid]/comm 空值	/proc/*/comm	非零告警（意味着进程被隐藏）
/proc/[pid]/exe 指向 deleted	/proc/*/exe	非零告警（攻击者替换二进制后）
内核模块数	/proc/modules	数量变化立刻告警

0.4.6.6 入侵检测的“误报”难题

异常画像的最大敌人是**误报**。下面这几条经验可以减少误报:

```
# 1. 建立业务基线（不要上来就告警）
#    跑 14 天的“观察模式”，记录正常波动范围

# 2. 用“滑动窗口”而非“瞬时值”告警
#    比如“连续 5 分钟 TCP 连接数 > 10000”而非“瞬时 > 10000”

# 3. 区分“系统进程”和“用户进程”
#    容器内的 PID 1-100 是系统进程，101+ 才是用户进程
#    重点关注后者的异常

# 4. 用白名单而非黑名单
#    列出合法进程名（sshd、nginx、postgres）→ 这些不告警
#    未知进程名 → 告警

# 5. 关联多指标
#    单一指标异常不一定告警
#    “TCP 连接数增加 + /proc/[pid]/comm 空 + 内核模块变化”三个一起出现 → 告警
```

0.4.6.7 完整可观测性 pipeline（从 /proc 到 Grafana）

```
/proc/[pid]/*       \
/proc/net/*         |    node_exporter（每分钟抓取）
/proc/sys/*         /    → Prometheus（存储）

auditd 日志         \
Falco 告警          |    → Loki / Elasticsearch（日志存储）
eBPF 输出          /    → Grafana（可视化）
                           ↓
                        Alertmanager（告警去重/路由）
                           ↓
                        企业微信（通知）
```

推荐技术栈：- **指标**：Prometheus + node_exporter + 自定义 exporter - **日志**：Loki（轻量）或 Elasticsearch（重）- **告警**：Falco（规则丰富）或 Tetragon（基于 eBPF）- **可视化**：Grafana - **通知**：Alertmanager + 企业微信

0.4.6.8 可观测性层的“代价”

- **auditd 全规则**会让 ps 工具产生海量日志，可能让 syslog 崩溃。建议从精细过滤开始。
- **Falco** 默认规则集可能产生 30% 误报，需要 1-2 周的“调参期”。
- **eBPF 程序**写错会让内核 panic，必须在 staging 环境验证。
- **指标存储**成本随集群规模线性增长。

推荐上线节奏：

```
# 第 1 周：部署 Prometheus + node_exporter
# 第 2 周：接入关键指标（进程数、TCP 连接数、CPU、内存）
# 第 3 周：部署 Falco，开启 default 规则集（audit 模式）
# 第 4 周：调参完成后切换到 enforce
# 第 5 周：部署 Grafana dashboard + Alertmanager
# 第 6 周：eBPF 自定义追踪（仅对关键路径）
```

业务方需要接受的现实：**没有 0 误报的入侵检测**。所有 IDS 都需要持续调参。把误报当“代价”，不要把它当“bug”。

0.4.7 实战案例：一家电商的 /proc 纵深防御 90 天落地

我们跟踪一家中等规模的电商公司（自营 + 三方店铺，订单峰值 5 万 QPS）从 0 到 1 完成 /proc 纵深防御的全过程。他们的起点很典型：

- 1000+ 台物理机和虚拟机（8 成都跑 CentOS 7 / Ubuntu 18.04 这种老内核）
- 安全团队只有 3 个人
- 没有专职 SRE，安全 / SRE 是同一拨人
- 上一季度刚出过一起内部挖矿事件，靠 top 偶然抓到

0.4.7.1 Day 0：基线盘点（不要上来就加固）

加固之前最重要的事是**搞清楚现状**。他们做的第一件事是写了一个盘点脚本：

```
#!/usr/bin/env bash
# /usr/local/bin/proc_baseline.sh
# 用途：基线盘点脚本，记录全集群 /proc 当前状态

set -euo pipefail
OUTDIR=/var/log/proc-baseline/$(date +%Y%m%d)
mkdir -p "$OUTDIR"

echo "[*] 内核版本与启动参数"
uname -a > "$OUTDIR/uname.txt"
cat /proc/cmdline > "$OUTDIR/cmdline.txt" 2>/dev/null || true

echo "[*] /proc 挂载选项"
findmnt /proc > "$OUTDIR/proc-mount.txt"
findmnt /proc/sys/fs/protected_* >> "$OUTDIR/proc-mount.txt" 2>/dev/null || true

echo "[*] 关键 sysctl"
sysctl -a | grep -E \
  "kernel.yama.ptrace_scope|kernel.randomize_va_space|kernel.kptr_restrict|\
kernel.dmesg_restrict|kernel.unprivileged_bpf_disabled|kernel.modules_disabled|\
kernel.usermodehelper|fs.protected_.*" \
  > "$OUTDIR/sysctl.txt"

echo "[*] AppArmor / SELinux 状态"
apparmor_status > "$OUTDIR/apparmor.txt" 2>/dev/null || echo "not installed"
sestatus > "$OUTDIR/selinux.txt" 2>/dev/null || echo "not installed"

echo "[*] /proc 暴露面：哪些路径对其他用户可读"
find /proc -maxdepth 3 -perm -o+r -type f 2>/dev/null \
  | grep -vE "/proc/(self|kthreadd|kworker|irq|softirq|migration)" \
  | head -200 > "$OUTDIR/world-readable-proc.txt"

echo "[*] 当前可观测组件清单"
systemctl list-unit-files --state=enabled 2>/dev/null \
  | grep -iE "prometheus|node_exporter|falco|auditd|tetragon|\
loki|grafana|alertmanager" \
  > "$OUTDIR/observability.txt" || true

echo "[*] 输出到 $OUTDIR"
```

跑完之后，他们盘点出的「典型老系统画像」大概是这样的：

```
1000 台机器的统计：
- 92% 没启用 hidepid
- 78% yama.ptrace_scope = 0（经典模式，可被同 uid ptrace）
- 65% kptr_restrict = 0（暴露内核符号）
- 43% 装了 AppArmor 但所有 profile 是 complain 模式
- 11% 启用了 auditd 但只监控 auth.log 相关规则
- 0%  部署了 Falco / Tetragon
- 87% /proc/sys/kernel/core_pattern 是默认的 "core"，没改成 |/bin/false
```

这张画像本身就是**对老板最好的汇报材料**——它用数字说明「我们现在的暴露面有多大」。安全预算从来不是靠说「我很重要」拿到的，是靠「现状盘点」拿到的。

0.4.7.2 Day 1-30：第 1 层 + 第 2 层（零停机可灰度）

第 1 个月他们只做两件事，因为这两件事**对业务零影响**：

1. **调整 sysctl**（重启前用 `sysctl -w` 在线生效，重启后写进 `/etc/sysctl.d/99-proc-hardening.conf` 持久化）

2. **改 systemd unit**，把 ProtectProc=invisible / PrivateProc= 等选项打开

他们写了一个**半自动化的灰度脚本**，按机房分批推：

```
#!/usr/bin/env bash
# /usr/local/bin/rollout-proc-hardening.sh
# 用途：分机房灰度推送 /proc 加固配置

set -euo pipefail
IDC="${1:?用法：$0 <idc-name>}"
CONF="/etc/sysctl.d/99-proc-hardening.conf"

case "$IDC" in
  idc-a) ROLLOUT_PCT=10 ;;   # 灰度起点
  idc-b) ROLLOUT_PCT=30 ;;
  idc-c) ROLLOUT_PCT=60 ;;
  idc-d) ROLLOUT_PCT=100 ;;  # 主要机房全量
  *) echo "unknown idc: $IDC"; exit 1 ;;
esac

echo "[*] 灰度比例：${ROLLOUT_PCT}%"

# 用 ansible / salt / puppet 都行，这里以 ansible 为例
ansible "idc_${IDC}_*" -m copy \
  -a "src=$CONF dest=$CONF mode=0644 owner=root group=root" \
  --percent "${ROLLOUT_PCT}"

ansible "idc_${IDC}_*" -m shell \
  -a "sysctl -p $CONF && systemctl daemon-reload" \
  --percent "${ROLLOUT_PCT}" --forks 20

# 观察 24h，没异常再推下一波
echo "[*] 等待 24h 后继续推下一波..."
```

这一阶段踩到的坑：

- **Logstash 挂了**：因为 hidepid=2 生效后，Logstash 用一个普通用户跑，它的子进程看不到其他 PID → 它自身的监控告警脚本也看不到目标进程 → 告警链路全断。**解法**：给 Logstash 加 SupplementaryGroups=systemd-journal 或者专门建一个 proc-monitor 组，把需要看全局 PID 的工具放进这个组。
- **监控系统自己被影响**：node_exporter 默认以 nobody 跑，开了 hidepid=2 后它采集不到 /proc/[pid]/stat → 进程指标全 0。**解法**：在 /etc/fstab 里把 gid=node_exporter 加到 proc 的挂载选项。
- **研发同事 ssh 进来 debug 找不到进程**：这是**预期内**的副作用，他们用「研发必须通过堡垒机登录、堡垒机用的是专门的低权限用户能看全局 PID」的方式解决。堡垒机不能在「加固」之后变成「研发的工作障碍」，否则所有人会想办法绕过它。

0.4.7.3 Day 31-60：第 3 层 + 第 4 层（灰度 + 回滚预案）

这个阶段上 AppArmor enforce + read-only bind mounts（针对 core_pattern 等）。**这两件事的破坏性更大**，必须准备好回滚：

```
#!/usr/bin/env bash
# /usr/local/bin/rollback-proc-hardening.sh
# 紧急回滚脚本，把加固配置全部还原

set -euo pipefail

echo "[!] 紧急回滚 /proc 加固"

# 1. 卸载所有 read-only bind mount
for mp in $(mount | awk '/\/run\/proc-ro/{print $3}'); do
  umount "$mp" || echo "[!] umount $mp 失败"
done
```

```
# 2. 还原 sysctl
rm -f /etc/sysctl.d/99-proc-hardening.conf
sysctl --system

# 3. AppArmor 切回 complain
aa-complain /etc/apparmor.d/* 2>/dev/null || true

# 4. 重启（最稳的回滚方式）
echo "[!] 即将重启以完成回滚（按 Ctrl+C 取消，10 秒后自动执行）"
sleep 10
systemctl reboot
```

这一阶段真正出过一个事故：某天早上 7 点，订单服务的 AppArmor profile 因为没覆盖到某个新引入的二进制（一个 Go 服务用 `cgo` 调了 `ptrace`），整个服务被 enforce 模式直接拒绝启动。订单瞬间掉零。

回滚用了 8 分钟（从告警到订单恢复），其中：

- 4 分钟：oncall 安全工程师 ssh 进去看 dmesg（他忘了 `dmesg_restrict=1`，幸好看得到）
- 2 分钟：手动 `aa-complain` → 改 profile → reload
- 1 分钟：服务起来
- 1 分钟：验证订单链路

事后他们做的三件事（这件事我每次讲都给团队重复）：

1. **所有 AppArmor profile 上线前必须在 staging 跑满 7 天**，7 天内的 deny 事件自动汇总成「应该加的 allow 规则」喂回去。
2. **oncall 手册里加一条**：「如果服务挂了且 dmesg 里看到 `audit: type=1400 audit: ... apparmor="DENIED"`，第一步永远是 `aa-complain`，不要去改 profile」。
3. **AppArmor 变更禁止在早高峰（8-10 点）和晚高峰（18-22 点）执行**。所有加固变更窗口固定在凌晨 2-5 点。

0.4.7.4 Day 61-90：第 5 层（可观测性 + 入侵检测）

前 60 天做的所有事，**没有一件事能告诉你「我被打了」**。所以最后 30 天他们上 Falco + Prometheus。

但上 Falco 之前他们做了一件很聪明的事：**先开 14 天 audit 模式**。也就是说，Falco 检测到的事件只入 Loki，不发告警。他们用这 14 天的日志做了一件所有团队都应该做但都懒得做的事——**统计误报率**：

```
14 天 audit 模式结果：
- 总事件：2,847,293
- 真阳性：47
- 误报率：99.998%
```

99.998% 误报。这就是 IDS 的现实。如果直接 enforce，业务方会在第一天就把告警 channel 静音。

第 15 天他们做了一次「白名单化」：把所有已知合法的「读 /proc/[pid]/environ」行为加白。误报率从 99.998% 降到 87%（仍然是大部分误报，但已经是合理范围）。剩下的 87% 误报都是「合规扫描器」「备份工具」「监控 agent」的合法行为，需要继续加白。

90 天后的最终状态：

```
- 100% 机器启用 hidepid=2
- 98% 机器 yama.ptrace_scope=1（剩下的 2% 是 Jenkins controller，给构建权限）
- 100% 机器启用 auditd + 自定义 /proc 规则
- 75% 服务上线 AppArmor enforce（剩下 25% 是历史包袱服务，慢慢迁）
- 100% 机器部署 node_exporter + Prometheus
- 60% 集群部署 Falco（剩下 40% 是测试 / 内网工具集群）
- 0 起已知的「靠 hook ps 隐身」的入侵事件
```

0 起是关键。不是说不可能，是说**他们用 90 天的时间，把攻击者最常用的隐身手法彻底废掉**。如果攻击者还想藏，他必须升级到 eBPF hook，那是另一个量级的成本——大多数挖矿脚本根本做不到。

0.4.7.5 经验与教训

做对的：1. **从盘点开始**——用数字说服老板，而不是用「直觉」2. **分阶段、灰度、回滚预案三件套**——任何加固措施上线前都要有这三件套 3. **前 60 天不告警**——只审计，只盘点，给团队磨合的时间 4. **回滚时间被严格控制**——8 分钟，从告警到恢复，这是演练的结果

做得不够好的：1. **早期没把「研发视角」纳入决策**——`hidepid=2` 让研发 debug 受影响，被绕过两次 2. **AppArmor profile 是手工写的**——应该用 `aa-genprof` 自动生成 3. **Falco 的规则集没和业务团队一起 review**——上线后才发现「订单服务的备份脚本会读所有 PID 的 environ」是合法行为 4. **没做压测**——auditd 全规则在峰值流量下让 syslog 排队了几个小时

给类似团队的建议：

> **90 天是底线**。任何想「两周搞定纵深防御」的方案，要么是过度承诺，要么是把代价藏在未来。

0.4.8 小结 & 练习

0.4.8.1 五层纵深防御的核心思想

回顾本章内容，我把 `/proc` 的纵深防御总结成五句话：

1. **减少暴露面**——让其他用户看不到的东西，攻击者也看不到
2. **访问控制**——谁能在什么条件下读 / 写哪些文件
3. **行为约束**——即使你能读 / 写，你也只能做合法的事
4. **完整性保护**——文件本身不被篡改
5. **可观测性**——任何异常都留痕、可查、可告警

这五层**不是替代关系，是叠加关系**。少任何一层，剩下的都会被绕过。

0.4.8.2 五层的「代价 / 收益」对照

层	收益	代价	适用场景
第 1 层：减少暴露面	高	低	**所有机器必须做**
第 2 层：访问控制	高	中	所有面向多用户的机器
第 3 层：行为约束	中-高	中-高	长期运行的关键服务
第 4 层：完整性保护	高	高	高安全要求 / 合规场景
第 5 层：可观测性	高	中	**所有机器必须做**

第 1 层和第 5 层是「无条件做」，其他三层根据业务需要选。

0.4.8.3 「好的 / 不好的」实战习惯

好的习惯：- 上线前用盘点脚本留 baseline - 加固前写好回滚脚本 - AppArmor / seccomp profile 在 staging 跑满 7 天再上生产 - 变更窗口固定在凌晨 2-5 点 - IDS 先 audit 模式 14 天再 enforce - 误报率统计 + 业务团队 review - 所有加固措施配套 oncall 手册

不好的习惯：- 一上来就 enforce AppArmor / seccomp - 用「应该不会出问题」代替回滚预案 - 让研发把堡垒机当成「麻烦」——说明设计有问题 - IDS 直接 enforce，告警 channel 一天内被静音 - 加固措施只在「出问题的那次」上线，事后又退回原状

0.4.8.4 自测题

题 1（基础）：你的生产机器上，攻击者拿到任意一个普通用户 shell 后，能读 `/proc/[pid]/environ` 吗？请用命令验证。

参考答案

```
# 在普通用户下试
cat /proc/1/environ
# 如果返回 Permission denied 或 No such file or directory → 你的 hidepid 配置生效了
# 如果返回一堆环境变量 → 你裸奔了
```

题 2（进阶）：你的机器上 yama.ptrace_scope 是几？换句话说，同 uid 的两个进程能互相 ptrace 吗？

参考答案

```
cat /proc/sys/kernel/yama/ptrace_scope
# 0 = classic（可以，攻击者能 attach 同 uid 进程 dump 内存）
# 1 = restricted（仅父子，不能）
# 2 = admin only（仅 CAP_SYS_PTRACE）
# 3 = no ptrace（完全禁用）
# 推荐 1 或 2
```

题 3（实战）：写一段 bash，从 /proc/net/tcp 还原出当前所有 ESTABLISHED 连接的「PID + 进程名 + 远端 IP:port」。要求能在你的机器上直接跑。

参考答案

```
#!/usr/bin/env bash
# 从 /proc/net/tcp 还原所有 ESTABLISHED 连接
set -euo pipefail

# 1. /proc/net/tcp 第 4 列 st=01 表示 ESTABLISHED
# 2. 第 10 列是 inode
# 3. 从 inode 反查 /proc/*/fd/* 找到 PID

while read -r inode; do
  for pid in /proc/[0-9]*; do
    if ls -l "$pid/fd/" 2>/dev/null | grep -q "socket:\[$inode\]"; then
      pid_num=$(basename "$pid")
      comm=$(cat "$pid/comm" 2>/dev/null)
      echo "inode=$inode pid=$pid_num comm=$comm"
      break
    fi
  done
done < <(awk '$4=="01" {print $10}' /proc/net/tcp)
```

进阶版（远端 IP 反查域名）：

```
# 上面那个脚本里，从 /proc/net/tcp 第 3 列 rem_address 提取
# 格式是 hex_ip:hex_port，例如 5D9CXXXX:01BB
# 反查域名用：getent hosts $(printf '%d.%d.%d.%d' 0x5D 0x9C 0xXX 0xXX)
```

题 4（架构）：你的团队要做 /proc 纵深防御，安全团队 3 人，没有专职 SRE，业务不能停。**你会怎么排 90 天计划**？请写出每一周的关键交付物。

参考答案

这个题的答案不唯一，但有几个关键节点必须出现：

- 第 1 周：盘点 + baseline 留底（必须先盘点，再加固）
- 第 2 周：sysctl 调整（零停机，灰度 10% → 30% → 60% → 100%）
- 第 3 周：hidepid=2 上线 + 回滚预案演练
- 第 4 周：AppArmor / seccomp 在 staging 起步（生产不上）
- 第 5-6 周：AppArmor / seccomp 灰度上生产
- 第 7 周：auditd 全规则上线
- 第 8 周：Falco audit 模式（不告警，只入日志）
- 第 9 周：Falco 误报统计 + 业务团队 review + 白名单化
- 第 10 周：Falco enforce + Grafana dashboard
- 第 11-12 周：调优 + oncall 手册 + 红蓝对抗演练

题 5（思考）：为什么「IDS 的告警不可能 0 误报」？从信息论角度解释。

参考答案

从信息论角度：**任何「正常」的定义都是基于历史数据的统计推断**。

- 攻击者只要让自己的攻击行为在统计上「看起来像正常行为」(low-and-slow、模仿合法进程名、利用业务高峰时段)，就能绕过基于统计的检测

- 防御者必须基于「已知正常」建模，但「已知正常」永远是不完备的
- 所以「正常 / 异常」是**近似**的二分，不是真正的二分
- 误报率只能逼近 0，不能等于 0

这是**所有 IDS 的根本限制**，不只是 Falco 或 Tetragon。这个限制告诉我们：

- 不要追求「0 误报」，要追求「误报可控 + 告警响应及时」
- 告警系统的设计核心是「让人愿意看告警」，不是「让告警完美」

0.5 第 4 章：可观测性落地——从一行 cat 到一套生产级告警体系

第 3 章讲了五层防御的「应然」——理论上怎么保护 /proc。但理论和生产之间隔着一段**很长的路**。这一章我们走一遍「从一行命令到一套生产级告警体系」的完整落地过程。

0.5.1 为什么”可观测性”是单独一章

/proc 本身就是一个**天然的观测点**——它把内核里关于进程、内存、网络、设备的一切都暴露成文件。理论上你只需要：

```
cat /proc/net/tcp
```

就能看到所有 TCP 连接。问题是：**怎么把这一行命令的输出变成可用的告警**？

- 你不可能让一个人 24 小时盯着 cat 输出
- 你也不能每秒钟跑一次 cat 然后 grep，那是 90 年代的办法
- 你需要的是「自动化采集 + 结构化存储 + 规则引擎 + 告警分发」——也就是可观测性体系

这一章我们重点讲三件事：

1. **指标采集**：从 /proc 到 Prometheus（4.2-4.4）
2. **日志采集**：从 auditd 到 Loki（4.5-4.6）
3. **告警调优**：从「满屏误报」到「精确告警」（4.7-4.9）

0.5.2 /proc 暴露面：指标准备度评估

在动手部署任何采集器之前，**先盘点你机器上的 /proc 暴露面**。这一步和第 3.6.1 节的 baseline 类似，但更聚焦于「采集器能不能拿到它需要的东西」。

```
#!/usr/bin/env bash
# /usr/local/bin/proc_observability_readiness.sh
# 用途：评估当前机器 /proc 对可观测性工具的友好度

set -euo pipefail
echo "=== /proc 可观测性准备度评估 ==="
echo ""

score=0
max=0

check() {
  local desc="$1"
  local cmd="$2"
  local expected="$3"
  max=$((max + 1))

  if result=$(bash -c "$cmd" 2>&1); then
    if echo "$result" | grep -qE "$expected"; then
      echo "[✓] $desc"
      score=$((score + 1))
    else
      echo "[☒] $desc"
      echo "    期望匹配: $expected"
```

```
      echo "    实际输出: $result"
    fi
  else
    echo "[☒] $desc (命令失败)"
  fi
}

# 1. node_exporter 能读 /proc/*/stat 吗?
check "node_exporter 可读 /proc/1/stat" \
  "sudo -u nobody cat /proc/1/stat 2>&1 | head -c 50" \
  "^[0-9]+"

# 2. node_exporter 能读 /proc/net/tcp 吗?
check "node_exporter 可读 /proc/net/tcp" \
  "sudo -u nobody head -1 /proc/net/tcp 2>&1" \
  "sl.*local_address"

# 3. auditd 能监听 /proc/sys/kernel/core_pattern 吗?
check "audit 规则支持 -w /proc/sys" \
  "auditctl -l 2>/dev/null | grep -E 'core_pattern|/proc/'" \
  ".*"

# 4. 当前 machine-id 存在 (Prometheus 标签需要)
check "machine-id 可读" \
  "cat /etc/machine-id 2>/dev/null" \
  "^[a-f0-9]{32}$"

# 5. 当前系统时间同步 (Prometheus 抓取需要准确时间)
check " 时间同步在 1 秒以内" \
  "chronyc tracking 2>/dev/null | grep 'System time' | awk '{print \$4}' | awk -F'=' '{print \$2}' | awk '{ if (\$1+0 < 1) exit 0;
  ".*"

echo ""
echo "=== 得分: $score / $max ==="
if [ "$score" -lt 3 ]; then
  echo "[!] 准备度不足，建议先调整权限再上可观测性工具"
fi
```

跑完一遍你就知道**采集器能不能装上去**。如果得分低于 3，说明你的 `/proc` 加固过严了——采集器需要某些路径的可读权限才能工作。

0.5.3 /proc 暴露面与采集器的权限协调

这是一个**真实的痛点**：加固做过头，监控就抓瞎。

`hidepid=2` 让非 root 用户看不到其他 PID → node_exporter 抓不到 `/proc/[pid]/stat` → Prometheus 里的进程指标全 0。

正解不是「关掉 hidepid」，是「给采集器开例外」：

```
# /etc/fstab 里的 proc 行
proc  /proc  proc  defaults,hidepid=2,gid=procmon  0  0

# 建一个组，把所有采集器用户加进去
groupadd -r procmon
usermod -aG procmon node_exporter
usermod -aG procmon prometheus
usermod -aG procmon falco
```

`hidepid=2,gid=procmon` 的含义是：所有用户看不到 PID，但 `procmon` 组的成员除外。这样既保护了普通用户，又给采集器开了口子。

这个技巧是所有做 /proc 加固 + 可观测性的团队都会踩的坑。把它写在 oncall 手册的第一页。

0.5.4 node_exporter 的 /proc 采集深度定制

默认的 node_exporter 暴露了一堆 /proc 相关指标。但**默认配置不是生产配置**。我们来看几个常见的深度定制。

0.5.4.1 自定义 /proc/[pid]/io 采集

/proc/[pid]/io 记录每个进程的 I/O 计数（read_bytes、write_bytes）。这是诊断「为什么这个进程 I/O 高」的黄金指标。

但默认 node_exporter 只采集系统的整体 I/O，不采集每个进程的。打开它：

```
# /etc/prometheus/node_exporter.yml
--collector.proc
processes:
  - pid: 1
    name: "systemd"
  - pid: "self"
    name: "node_exporter"
```

或者用 --collector.processes + 单独的 procfs collector。但更优雅的做法是写一个**自定义 exporter**：

```
#!/usr/bin/env python3
# /usr/local/bin/proc_io_exporter.py
# 把 /proc/[pid]/io 导出为 Prometheus 指标

import os
import time
from prometheus_client import start_http_server, Gauge, Counter

PROC_IO_FIELDS = [
    "rchar", "wchar", "syscr", "syscw",
    "read_bytes", "write_bytes", "cancelled_write_bytes",
]

gauges = {}
for field in PROC_IO_FIELDS:
    gauges[field] = Gauge(
        f"proc_io_{field}_bytes_total",
        f"Per-process {field} from /proc/[pid]/io",
        ["pid", "comm"],
    )

def collect():
    for entry in os.listdir("/proc"):
        if not entry.isdigit():
            continue
        pid = entry
        io_file = f"/proc/{pid}/io"
        try:
            with open(io_file) as f:
                data = {}
                for line in f:
                    k, v = line.strip().split(": ")
                    data[k] = int(v)
            with open(f"/proc/{pid}/comm") as f:
                comm = f.read().strip()
        except (FileNotFoundError, ProcessLookupError, PermissionError):
            continue

        for field in PROC_IO_FIELDS:
            if field in data:
                gauges[field].labels(pid=pid, comm=comm).set(data[field])

if __name__ == "__main__":
```

```
    start_http_server(9878)
    while True:
        collect()
        time.sleep(5)
```

跑起来之后，Prometheus 端可以这样查：

```
# 找出 I/O 最高的 10 个进程
topk(10, sum by (comm) (rate(proc_io_write_bytes_total[5m])))

# 找出 I/O 异常突增的进程（与 1 小时前对比）
delta = sum by (comm) (proc_io_write_bytes_total - proc_io_write_bytes_total offset 1h)
delta > 1e9
```

0.5.4.2 把 /proc/net/tcp 状态计数变成 Prometheus 指标

/proc/net/tcp 里有所有 TCP 连接的状态。node_exporter 默认采集 node_sockstat_TCP_*，但粒度不够。我们想要的是「每个状态的连接数随时间变化曲线」，用来检测「突增的 ESTABLISHED 连接」（可能是 C2 通道）。

```
#!/usr/bin/env python3
# /usr/local/bin/proc_tcp_state_exporter.py
# 从 /proc/net/tcp 解析状态，导出 Prometheus 指标

import time
from prometheus_client import start_http_server, Gauge

TCP_STATE = {
    "01": "ESTABLISHED",
    "02": "SYN_SENT",
    "03": "SYN_RECV",
    "04": "FIN_WAIT1",
    "05": "FIN_WAIT2",
    "06": "TIME_WAIT",
    "07": "CLOSE",
    "08": "CLOSE_WAIT",
    "09": "LAST_ACK",
    "0A": "LISTEN",
    "0B": "CLOSING",
}

state_gauges = {
    name: Gauge(f"proc_tcp_state_{name.lower()}", f"Count of TCP in {name}")
    for name in TCP_STATE.values()
}

def parse_proc_net_tcp(path):
    counts = {name: 0 for name in TCP_STATE.values()}
    with open(path) as f:
        next(f)  # skip header
        for line in f:
            parts = line.split()
            if len(parts) < 4:
                continue
            st = parts[3]
            name = TCP_STATE.get(st)
            if name:
                counts[name] += 1
    return counts

def collect():
    for proto, path in [("v4", "/proc/net/tcp"), ("v6", "/proc/net/tcp6")]:
        counts = parse_proc_net_tcp(path)
        for name, count in counts.items():
```

```
            state_gauges[name].set(count)

if __name__ == "__main__":
    start_http_server(9879)
    while True:
        collect()
        time.sleep(10)
```

0.5.4.3 自定义指标的元数据

让所有自定义指标都带 **机器标签、机房标签、业务标签**：

```
# 在 exporter 启动时从环境变量读
import os

MACHINE_ID = open("/etc/machine-id").read().strip()
IDC = os.environ.get("IDC", "unknown")
SERVICE = os.environ.get("SERVICE", "unknown")

gauge = Gauge(
    "proc_io_bytes",
    "Per-process I/O",
    ["pid", "comm", "machine_id", "idc", "service"],
)
gauge.labels(pid=pid, comm=comm, machine_id=MACHINE_ID, idc=IDC, service=SERVICE).set(value)
```

这样 Prometheus 抓到的指标天然可以按机房 / 业务聚合。

0.5.5 auditd 的 /proc 规则深度配置

auditd 是 Linux 原生的审计子系统，能记录「谁在什么时候读了哪个文件」。第 3.5.4 节我们上了基础规则，这一节我们深入。

0.5.5.1 细粒度 /proc 监控规则

```
# /etc/audit/rules.d/10-proc-deep.rules

# --- /proc/[pid]/environ 读取（重点：凭证窃取）---
-a always,exit -F arch=b64 -S openat -F a2=0 -F path=/proc/[0-9]*/environ -F key=proc-environ-read

# --- /proc/[pid]/mem 读取（重点：内存取证 / 注入）---
-a always,exit -F arch=b64 -S process_vm_readv -F key=proc-mem-read
-a always,exit -F arch=b64 -S ptrace -F a0=0x4 -F key=proc-ptrace-read

# --- /proc/[pid]/maps 读取（重点：内存布局发现）---
-a always,exit -F arch=b64 -S openat -F path=/proc/[0-9]*/maps -F key=proc-maps-read

# --- /proc/[pid]/fd/ 目录读取（重点：socket 反查）---
-a always,exit -F arch=b64 -S getdents64 -F dir=/proc/[0-9]*/fd -F key=proc-fd-list

# --- /proc/sys/kernel/core_pattern 写入（重点：容器逃逸 CVE-2025-52881 类）---
-w /proc/sys/kernel/core_pattern -p wa -k proc-core-pattern-write

# --- /proc/sys/kernel/modules_disabled 写入（重点：禁用模块加载开关）---
-w /proc/sys/kernel/modules_disabled -p wa -k proc-modules-write

# --- /proc/kallsyms 读取（重点：内核符号探测）---
-a always,exit -F arch=b64 -S openat -F path=/proc/kallsyms -F key=proc-kallsyms-read

# --- /proc/net/tcp 读取（重点：连接发现，但有大量正常读取，需要白名单）---
-a always,exit -F arch=b64 -S openat -F path=/proc/net/tcp -F key=proc-net-tcp-read
-a always,exit -F arch=b64 -S openat -F path=/proc/net/tcp6 -F key=proc-net-tcp-read
```

0.5.5.2 减少 auditd 性能开销的关键

auditd 全规则会让 ps / top 慢 3-5 倍。**生产环境的真实情况是：默认规则会让 CPU 占用飙升。**

优化 1：用 -F pid!= 排除监控 agent 自己

```
# node_exporter 的 PID 是动态的，启动时记录下来
NODE_EXPORTER_PID=$(pgrep -f node_exporter | head -1)

# 把这个 PID 加到排除列表
-a never,exit -F pid=$NODE_EXPORTER_PID
```

优化 2：用 buffer-size 调大缓冲

```
# /etc/audit/auditd.conf
buffer_size = 8192    # 默认 1024 太小
backlog_limit = 8192
failure_mode = 1      # 1 = printk（不阻塞）
```

优化 3：用 key 字段做精细过滤

不要对所有 /proc/* 读都监控。聚焦在**敏感路径**：

```
# 只监控敏感路径，普通路径不监控
-a always,exit -F path=/proc/[0-9]*/environ
-a always,exit -F path=/proc/[0-9]*/mem
# 不要监控 /proc/[pid]/stat（太频繁，没价值）
```

0.5.5.3 auditd 日志解析：把二进制事件变结构化数据

auditd 输出的是**空格分隔的 key=value 文本**，但字段很多，手工解析麻烦。用 ausearch + aureport 是基本功：

```
# 查所有读 /proc/[pid]/environ 的事件
ausearch -k proc-environ-read --interpret | head -20

# 查所有写 core_pattern 的事件
ausearch -k proc-core-pattern-write --interpret

# 统计昨天的 proc-environ-read 事件数量
aureport --start yesterday --end now --key proc-environ-read

# 按进程统计
ausearch -k proc-environ-read --interpret | \
  awk '/comm=/{comm=$0} /exe=/{print comm, $0}' | \
  sort | uniq -c | sort -rn | head
```

更进一步，把 auditd 日志**实时结构化**到 Loki / Elasticsearch：

```
# /etc/audit/plugins.d/au-remote.conf
active = yes
direction = out
path = /sbin/au-remote
type = always
```

或者用 audit-log-parser 这种自写工具，把每条事件转成 JSON 推到 Kafka。

0.5.6 Loki + Promtail：/proc 异常的实时日志聚合

auditd 日志量巨大（高峰期每分钟 10 万条），不适合直接进 Elasticsearch。Loki 是更轻量的选择——它不索引原始文本，只索引标签。

0.5.6.1 Promtail 配置：抓 auditd 日志

```
# /etc/promtail/config.yml
server:
  http_listen_port: 9080
```

```yaml
positions:
  filename: /var/lib/promtail/positions.yaml

clients:
  - url: http://loki:3100/loki/api/v1/push

scrape_configs:
  - job_name: auditd
    static_configs:
      - targets:
          - localhost
        labels:
          job: auditd
          __path__: /var/log/audit/audit.log
    pipeline_stages:
      # 1. 解析 auditd 格式
      - match:
          selector: '{job="auditd"}'
          stages:
            - regex:
                expression: 'type=(?P<type>\w+) msg=audit\((?P<ts>\d+.\d+):(?P<seq>\d+)\): (?P<rest>.*)'
            - labels:
                type:

      # 2. 提取 key= 字段
      - match:
          selector: '{job="auditd"}'
          stages:
            - regex:
                expression: 'key="(?P<key>[^"]+)"'
            - labels:
                key:

      # 3. 提交流到 Loki
```

0.5.6.2 实时查询：哪个进程读了 /proc/[pid]/environ

```
{job="auditd"} |= "proc-environ-read"
```

或者更精细：

```
{job="auditd"} |~ "key=\"proc-environ-read\"" | line_format `{{.entry}}`
```

进阶：检测「同一 PID 在 1 分钟内读 N 个其他 PID 的 environ」（这是攻击行为的标志）：

```
sum by (pid) (
  count_over_time(
    {job="auditd"} |= "key=\"proc-environ-read\"" [1m]
  )
) > 10
```

0.5.7 告警调优的核心思想

指标和日志都齐了，下一步是「告警」。

告警系统有一个**永恒的矛盾**：

- 告警太敏感 → 业务方静音告警 channel → 真出了事没人看
- 告警太迟钝 → 真出事的时候没有告警 → 攻击持续数小时才被发现

我的经验是：**告警的第一目标是「人愿意看」，第二目标才是「覆盖更多场景」。**

0.5.7.1 告警调优的三个阶段

阶段 1：audit 模式（14 天）

所有告警规则**不实际触发告警**，只入日志 / 入 metric。用这 14 天做：

- 统计每条规则的命中率
- 区分「真阳性」vs「误报」
- 标记需要白名单化的合法行为

阶段 2：白名单化（7 天）

把阶段 1 识别出的合法行为加白。比如：

- 监控 agent（node_exporter、Prometheus）读 `/proc` → 全加白
- 备份工具读所有 PID 的 cmdline → 加白（但只对 backup 用户）
- 合规扫描器读 `/proc/[pid]/environ` → 加白（但仅工作时间）

阶段 3：灰度 enforce（30 天）

按业务灰度，从「非关键业务」开始 enforce，关键业务最后 enforce。每个 enforce 后的 48 小时密切观察告警量。

0.5.7.2 「白名单」的正确写法

白名单不是「加个 `if comm == 'node_exporter'` 就行」。**生产环境的白名单必须基于「标签 + 时间窗口 + 频率」**：

```
# Prometheus Alertmanager 例
- alert: ProcEnvironRead
  expr: |
    sum by (exe, uid) (
      rate(audit_proc_environ_read_total[5m])
    ) > 0.1
  for: 5m
  labels:
    severity: warning
  annotations:
    summary: " 进程 {{ $labels.exe }} (uid={{ $labels.uid }}) 持续读取 /proc/[pid]/environ"
  # 白名单：node_exporter / prometheus / 备份用户
  # 但通过 alertmanager 的 inhibit_rules 实现
```

`inhibit_rules`：

```
inhibit_rules:
  - source_match:
      exe: 'node_exporter'
    target_match:
      exe: 'node_exporter'
    equal: ['exe']
```

但**这种白名单是脆弱的**——攻击者可以把自己的进程叫 `node_exporter`。**更强的白名单是用「绝对路径 + 文件所有者 + 签名」**：

```
# 白名单应该基于 exe 的 inode + 校验和
node_exporter_path=$(readlink -f /proc/$(pgrep node_exporter)/exe)
node_exporter_sha256=$(sha256sum "$node_exporter_path" | awk '{print $1}')
echo " 白名单：$node_exporter_path ($node_exporter_sha256)"
```

如果 inode 或 sha256 变了 → 进程被替换 → 白名单失效 → 触发告警。

0.5.8 告警模板：可直接套用的 PromQL / LogQL 规则集

下面这套规则集是**经过多家公司生产验证**的，可以直接用：

0.5.8.1 进程异常

```
# 异常进程数激增（可能是 fork bomb 或 rootkit）
expr: |
```

```
  delta(node_processes_state[5m]) > 100
for: 5m
labels: { severity: warning }

# 单进程 fd 数量异常（可能是被注入 socket）
expr: |
  count by (pid, comm) (
    proc_fd_count_total > 200
  ) > 0
for: 10m
labels: { severity: warning }
annotations:
  summary: "{{ $labels.comm }} (pid {{ $labels.pid }}) fd 数量超过 200"

# 进程 /proc/[pid]/comm 为空（隐藏进程）
expr: |
  proc_comm_empty == 1
for: 1m
labels: { severity: critical }
```

0.5.8.2 网络异常

```
# ESTABLISHED 连接数激增
expr: |
  delta(proc_tcp_state_established[5m]) > 1000
for: 5m
labels: { severity: warning }

# LISTEN 端口异常增多（攻击者开监听端口）
expr: |
  delta(proc_tcp_state_listen[1h]) > 5
for: 10m
labels: { severity: warning }
```

0.5.8.3 /proc 访问异常

```
# 单进程读 environ 频率异常
sum by (pid, exe) (
  count_over_time(
    {job="auditd"} |= "key=\"proc-environ-read\"" [5m]
  )
) > 50

# 非 root 用户读其他进程 maps（提权准备）
{job="auditd"} |= "key=\"proc-maps-read\"" | uid!=0
```

0.5.9 「告警疲劳」的工程化缓解

告警疲劳是真实的安全威胁。MITRE ATT&CK 的 T1562（Impair Defenses）明确把「让告警失效」列为攻击手段。攻击者**希望**你疲劳、希望你静音、希望你不看告警。

0.5.9.1 告警分级：P0 / P1 / P2 / P3

P0（critical，立即处理）：- `core_pattern` 被写入 - 非 root 进程读其他进程 `mem` - 内核模块加载 - 异常 `/proc/kallsyms` 读取（kptr_restrict=0 时）

P1（warning，30 分钟内）：- 单进程 fd 数量异常 - `/proc/[pid]/environ` 高频读取 - LISTEN 端口异常增加

P2（info，每日 review）：- `/proc/net/tcp` 读取频率（业务正常，但需观察）- 新进程创建（baseline 内可忽略）

P3（debug，每周 review）：- 任何 audit 事件全量入 Loki，仅供查询

0.5.9.2 Alertmanager 的抑制与去重

```yaml
# /etc/alertmanager/alertmanager.yml
route:
  receiver: 'default'
  group_by: ['alertname', 'idc']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
    - match:
        severity: critical
      receiver: 'pagerduty'
    - match:
        severity: warning
      receiver: 'wecom'

inhibit_rules:
  # P1 抑制 P2
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal: ['idc', 'alertname']

  # 同 alertname 在 5 分钟内只发一次
  - source_match:
      alertname: 'ProcEnvironRead'
    target_match:
      alertname: 'ProcEnvironRead'
    equal: ['pid']
```

0.5.9.3 告警接收人的轮值

最关键的一条：oncall 必须轮值。

很多团队的安全告警 channel 是个「公共 channel」，谁看到谁处理。**这是错的。**它意味着：

- 「反正不是我 oncall，不看」
- 「这条 alert 别人会处理吧」
- 「这条 alert 24 小时还没人理，估计不重要」

正解是：

```yaml
# 8 个人轮值，每人一周
receivers:
  - name: 'oncall-rotation'
    webhook_configs:
      - url: 'https://oncall-bot.internal/api/notify'
        send_resolved: true
```

告警落到**当周 oncall 个人**，不是落到「公共 channel」。oncall 必须在 30 分钟内 ack，否则升级到 manager。

0.5.10 小结

可观测性落地是个**系统工程**：

- 它不是「装一个 Prometheus」就完事
- 它是「采集 → 存储 → 规则 → 告警 → 接收 → 响应」的完整链路
- 链路里任何一环断了，告警就没意义

最重要的认知：告警系统的第一目标是「人愿意看」，第二目标才是「覆盖更多场景」。一个 30% 误报率但 oncall 愿意看的告警系统，比一个 0% 误报率但 oncall 静音的告警系统有用 100 倍。

下一章我们讲 /proc 的下一代——eBPF。eBPF 让你**不通过 /proc 也能观测进程**，这是 /proc 这套 1992 年接口的演化方向。

0.6 第 5 章：eBPF 与 /proc 的下一代——为什么内核社区在「绕过」/proc

写到这里，我想换个角度：之前所有的章节都在讲「怎么用 /proc」。但如果你关注 Linux 内核社区近 5 年的演进方向，你会发现一个有趣的趋势——**内核社区正在「绕过」/proc**。

0.6.1 /proc 的设计原罪：copy_to_user 的代价

/proc 的核心实现是「内核态数据 → 用户态文件」的翻译。每个 /proc/[pid]/stat 读取都是一次 copy_to_user 调用。对单次读取，这个代价很小。但当你**每分钟抓取 1000 个进程的 stat**，你就在内核和用户态之间搬运了 60 万次数据。

更糟的是，**/proc 的数据是「快照」**——你读 /proc/[pid]/stat 那一瞬间的值，不代表下一秒的值。如果攻击者在两次读取之间篡改了某个字段，你看到的是「篡改后」的状态。

这两个问题驱动了 eBPF 的崛起：**eBPF 程序运行在内核态，能直接访问内核数据结构，不通过 copy_to_user 翻译；eBPF 程序能持续运行，捕获「事件流」而不是「快照」**。

0.6.2 eBPF 是什么：30 秒入门

eBPF（extended Berkeley Packet Filter）是一种**内核虚拟机**。用户写一段 C 代码（或 Rust），编译成 eBPF 字节码，**安全地**加载到内核。eBPF 程序在内核态运行，能：

- 监听内核函数调用（kprobe）
- 监听用户态函数调用（uprobe）
- 监听网络包（xdp、tc）
- 监听系统调用（tracepoint）
- 修改网络包 / 系统调用参数

eBPF 的核心承诺是「**安全 + 性能 + 灵活**」：

- **安全**：eBPF 程序运行前被 verifier 验证，不会让内核 panic
- **性能**：JIT 编译后接近原生代码速度
- **灵活**：能 hook 内核几乎任何位置

0.6.3 eBPF vs /proc：核心差异

维度	/proc	eBPF
数据访问方式	用户态文件读取	内核态事件回调
数据延迟	快照（采样间隔）	实时（事件触发）
性能影响	取决于读取频率	接近零
攻击者可见	可见（/proc 是接口）	不可见（在内核中）
攻击者绕过难度	低（hook getdents 即可）	极高（需要 root + 内核 exploit）
防御者可用	是	是
攻击者可用	是（rootkit 主流手法）	是（BPF rootkit 已出现）

0.6.4 eBPF 怎么观测进程：四个 hook 点

eBPF 观测进程有四种主要 hook 点：

0.6.4.1 tracepoint：稳定但功能受限

tracepoint 是内核开发者**预定义**的 hook 点。比如 sched_process_exec（进程创建）、sched_process_exit（进程退出）。

```
// 监控所有新进程创建
SEC("tracepoint/sched/sched_process_exec")
int handle_exec(struct trace_event_raw_sched_process_exec *ctx) {
    char filename[256];
    bpf_probe_read_str(filename, sizeof(filename), ctx->filename);
    bpf_printk("exec: %s\\n", filename);
    return 0;
}
```

0.6.4.2 kprobe：灵活但版本敏感

kprobe 可以 hook **任意**内核函数。比如你想监控 `do_filp_open`（文件打开）：

```
SEC("kprobe/do_filp_open")
int handle_open(struct pt_regs *ctx) {
    char filename[256];
    bpf_probe_read_str(filename, sizeof(filename),
                       (char *)PT_REGS_PARM2(ctx));
    if (strstr(filename, "/proc/")) {
        bpf_printk("/proc access: %s\\n", filename);
    }
    return 0;
}
```

kprobe 的问题是：内核版本变了，函数签名可能变，`do_filp_open` 在 6.x 里变成 `do_filp_open` 内联或被替换。所以 kprobe 程序**必须随内核升级同步维护**。

0.6.4.3 uprobe：监控用户态函数

uprobe 让 eBPF 能 hook 用户态函数。比如监控 nginx 处理请求的入口：

```
SEC("uprobe/usr/sbin/nginx:ngx_http_process_request")
int handle_request(struct pt_regs *ctx) {
    bpf_printk("nginx request\\n");
    return 0;
}
```

0.6.4.4 LSM: Linux Security Module hook

LSM 是 Linux 安全框架的 hook 点。eBPF 通过 BPF LSM 可以拦截「权限检查」：

```
SEC("lsm/file_open")
int bpf_file_open(struct file *file) {
    char path[256];
    bpf_probe_read_str(path, sizeof(path), file->f_path.dentry->d_name.name);
    if (strstr(path, "/proc/")) {
        // 拦截 /proc 访问
        return -EPERM;  // 拒绝
    }
    return 0;
}
```

这是**最强大也最危险**的 eBPF hook 点——它能直接在内核权限检查路径上做决策。

0.6.5 实操：用 bcc 写一个 /proc 访问监控器

bcc（BPF Compiler Collection）让 eBPF 编程简单很多。我们写一个监控所有 `/proc` 访问的工具：

```
#!/usr/bin/env python3
# /usr/local/bin/proc_access_tracer.py
# 用 eBPF 监控所有读 /proc/[pid]/* 的访问

from bcc import BPF
```

```
prog = r"""
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
#include <linux/fs.h>

struct data_t {
    u32 pid;
    u32 uid;
    char comm[TASK_COMM_LEN];
    char filename[256];
};

BPF_HASH(events, u32, struct data_t, 1024);
BPF_PERF_OUTPUT(trace_events);

int kprobe__vfs_read(struct pt_regs *ctx, struct file *file,
                     char __user *buf, size_t count, loff_t *pos) {
    char filename[256] = {};
    bpf_probe_read_str(filename, sizeof(filename),
                       file->f_path.dentry->d_name.name);

    // 父目录名
    char parent[256] = {};
    bpf_probe_read_str(parent, sizeof(parent),
                       file->f_path.dentry->d_parent->d_name.name);

    // 只关心 /proc/[pid]/* 形式的访问
    // parent 可能是 "1", "1234" 等 PID
    // 且父目录的父目录是 "proc"
    char grandparent[256] = {};
    bpf_probe_read_str(grandparent, sizeof(grandparent),
                       file->f_path.dentry->d_parent->d_parent->d_name.name);

    if (grandparent[0] == 'p' && grandparent[1] == 'r' &&
        grandparent[2] == 'o' && grandparent[3] == 'c' && grandparent[4] == 0) {
        struct data_t data = {};
        u64 pid_tgid = bpf_get_current_pid_tgid();
        data.pid = pid_tgid >> 32;
        data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;
        bpf_get_current_comm(&data.comm, sizeof(data.comm));
        bpf_probe_read_str(data.filename, sizeof(data.filename), filename);

        trace_events.perf_submit(ctx, &data, sizeof(data));
    }

    return 0;
}
"""

b = BPF(text=prog)
b.attach_kprobe(event="vfs_read", fn_name="kprobe__vfs_read")

def print_event(cpu, data, size):
    event = b["trace_events"].event(data)
    print(f"pid={event.pid} uid={event.uid} comm={event.comm.decode()} "
          f"file={event.filename.decode()}")

b["trace_events"].open_perf_buffer(print_event)

print("Tracing /proc access... Ctrl-C to stop")
while True:
    b.perf_buffer_poll()
```

跑起来之后，你会看到类似这样的输出：

```
pid=1234 uid=1000 comm=bash file=environ
pid=1234 uid=1000 comm=bash file=cmdline
pid=5678 uid=0 comm=node_exporter file=stat
pid=9012 uid=1000 comm=curl file=maps
```

这比 auditd 的优势：

- 延迟：从「事件发生」到「输出」是微秒级
- 性能：eBPF 程序在内核态，不进用户态
- 隐蔽：攻击者看不到 auditd 日志，但 eBPF 程序他们也关不掉（除非 unpin + 卸载）

0.6.6 实操：用 bpftrace 写一行 /proc 监控

bpftrace 是 eBPF 的高级 DSL，比 bcc 更简洁。一行命令监控所有 /proc/[pid]/environ 读取：

```
bpftrace -e '
kprobe:vfs_read {
  $file = (struct file *)arg0;
  $dentry = $file->f_path.dentry;
  $name = $dentry->d_name.name;
  $parent = $dentry->d_parent->d_name.name;
  $grand = $dentry->d_parent->d_parent->d_name.name;

  if ($grand == "proc" && str($name, "environ")) {
    printf("pid=%d comm=%s read /proc/[%s]/%s\\n",
           pid, comm, $parent, $name);
  }
}
'
```

bpftrace 的强大在于**临时调试**——你不用编译、不用部署、不用维护，shell 里一行就能跑。

0.6.7 eBPF 在可观测性领域的生产实践

0.6.7.1 Cilium / Tetragon

Cilium 是基于 eBPF 的容器网络方案。Tetragon 是 Cilium 团队的安全可观测性产品。

```yaml
# Tetragon TracingPolicy
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: monitor-proc-access
spec:
  kprobes:
    - call: "security_file_open"
      syscall: false
      args:
        - index: 0
          type: "nop"
      selectors:
        - matchArgs:
            - index: 0
              operator: "Equal"
              values:
                - "/proc/[0-9]*/environ"
          matchActions:
            - action: Sigkill
              argError: -1
            - action: Post
              argError: -1
```

Tetragon 能在 hook 点上直接执行 Sigkill——检测到可疑访问时直接杀死进程。这是 auditd 做不到的。

0.6.7.2 Falco + eBPF

Falco 从 0.32 版本开始默认用 eBPF 作为驱动（之前的版本用 kernel module）。这让 Falco 的检测能力大大增强。

```
# /etc/falco/falco_rules.local.yaml
- rule: Read Sensitive File Untrusted
  desc: >
    Detect any process reading /proc/[pid]/environ except known agents
  condition: >
    open_read
    and proc.name != node_exporter
    and proc.name != prometheus
    and container.id != host
    and fd.name = "/proc/[0-9]*/environ"
  output: >
    Sensitive file read by untrusted process
    (user=%user.name command=%proc.cmdline file=%fd.name)
  priority: WARNING
  tags: [process, procfs, mitre_T1552]
```

0.6.7.3 Pixie（New Relic）

Pixie 是 New Relic 开源的 eBPF 可观测性产品，能自动发现「哪些进程读了哪些文件」「网络延迟在哪」「TLS 握手失败原因」等。

```
# Pixie 的 pxl 脚本
import px

# 查询「所有读 /proc/[pid]/environ 的进程」
df = px.DataFrame('proc_events', start_time='-5m')
df = df[df['event_type'] == 'open']
df = df[df['filename'] =~ '/proc/[0-9]+/environ']
px.display(df)
```

0.6.8 eBPF 在安全攻防中的双刃剑

eBPF 不仅防御者能用，攻击者也能用。这就是「双刃剑」。

0.6.8.1 BPF rootkit

2022 年起，公开的 BPF rootkit 出现：

- **Symbiote**（2022）：用 BPF 修改系统调用返回值，让 `ps`、`ls`、`netstat` 看不到攻击痕迹
- **BPFDoor**（2023）：用 BPF socket filter 监听特定端口，无需任何进程监听
- **ebpf-mapper**（2024）：用 BPF 修改网络包，绕过 IDS

这些 BPF rootkit 比传统 LKM rootkit 更难检测——因为：

- 它们不依赖内核模块（不需要加载.ko 文件）
- 它们在内核的 BPF 子系统里运行，常规 `lsmod` 看不到
- 它们 hook 在系统调用之前，auditd / `/proc` 看不到

0.6.8.2 检测 BPF rootkit 的方法

```
# 1. 列出所有 BPF 程序
bpftool prog show

# 2. 列出所有 BPF map
bpftool map show

# 3. 查看每个 BPF 程序的指令数（异常多可能是 rootkit）
```

```
bpftool prog show | awk '{print $1, $7}'  # prog_id, insns

# 4. 查看 BPF 子系统统计
cat /proc/sysctl/net.core.bpf_*  # 某些内核版本

# 5. 比对 BPF 程序数量预期值
# 比如：正常服务跑 50 个 BPF 程序，如果突然变成 200 → 有 rootkit
```

更强的检测：eBPF 程序被加载后会留在 BPF 子系统里（pin 到 bpffs）。检测脚本定期比对 BPF 程序列表的 hash：

```
#!/usr/bin/env bash
# /usr/local/bin/bpf_integrity_check.sh
# 用途：检测 BPF 子系统是否被植入 rootkit

EXPECTED_HASH_FILE="/var/lib/bpf-baseline/progs.sha256"
CURRENT_HASH=$(bpftool prog show 2>/dev/null | sha256sum | awk '{print $1}')

if [ -f "$EXPECTED_HASH_FILE" ]; then
    EXPECTED=$(cat "$EXPECTED_HASH_FILE")
    if [ "$CURRENT_HASH" != "$EXPECTED" ]; then
        echo "[!] BPF programs changed! Potential rootkit."
        echo "[!] diff: $EXPECTED → $CURRENT_HASH"
        # 进一步：列出新增的 BPF 程序
        diff <(echo "$EXPECTED") <(echo "$CURRENT_HASH")
    fi
else
    echo "$CURRENT_HASH" > "$EXPECTED_HASH_FILE"
    echo "[*] BPF baseline created"
fi
```

0.6.8.3 防御者用 BPF，攻击者用 BPF，怎么破?

答案是：双方都用 BPF，但防御者用更多 BPF。

具体策略：

1. **保护 BPF 子系统**：kernel.unprivileged_bpf_disabled=2 禁止非特权用户加载 BPF
2. **监控 BPF 子系统**：上面那个完整性检查脚本
3. **BPF LSM**：在内核权限检查路径上加 BPF hook，让非授权 BPF 程序加载直接被拒
4. **BPF tokenization**：未来特性——每个 BPF 程序有个 token，加载时验证 token 完整性

0.6.9 /proc 的未来：内核社区的演化方向

读 Linux 内核的 mailing list，能看到 /proc 的演化方向：

0.6.9.1 /proc 正在被「分层」

传统 /proc 是「一切都在一个文件系统」。新内核开始把 /proc 拆成多层次：

- /proc/self/ —当前进程
- /proc/thread-self/ —当前线程
- /proc/[pid]/task/[tid]/ —进程的特定线程
- /proc/[pid]/attr/ —安全上下文

未来：/proc 可能进一步拆分，比如：- /proc/cpuinfo → 拆到 /proc/cpu/0/info - /proc/meminfo → 拆到 /proc/memory/global - /proc/net/tcp → 拆到 /proc/net/tcp/v4、/proc/net/tcp/v6

0.6.9.2 新的内核观测接口

近 5 年内核社区引入的新接口：

- **/sys/kernel/tracing/** —tracefs，比 debugfs 更可控
- **/proc/pressure/** —PSI（Pressure Stall Information），CPU/内存/IO 压力指标

- **/proc/interrupts 分级** —按 NUMA node、按 CPU 拆分
- **cgroup v2** —取代 cgroup v1，提供更细粒度的资源控制
- **/sys/fs/bpf/** —BPF filesystem，让 BPF 程序可以 pin 在文件系统

0.6.9.3 /proc 不会被淘汰

虽然有 eBPF、cgroup v2 等新接口，/proc **不会消失**。原因：

- 兼容性：几十年的兼容性是不能破坏的
- 易用性：cat /proc/net/tcp 比写 eBPF 程序简单太多
- 标准化：/proc 接口已经被 POSIX 部分采纳

/proc 会演化成「稳定的、易用的、低开销的」接口——让普通用户能 5 秒内看到「TCP 连接列表」，而 eBPF 在底层支撑更高级的实时观测。

0.6.10 小结

eBPF 不是 /proc 的替代品，是 /proc 的**升级**：

- /proc 给普通用户**易用的快照**
- eBPF 给专业用户**高性能的事件流**

未来 5 年，/proc 的角色会越来越像「调试接口」，而生产环境的可观测性会全面转向 eBPF。但这不意味着 /proc 不重要——它依然是：

- 排查问题的第一站（cat /proc/[pid]/status）
- 应急响应的最后一道防线（即使所有监控挂了，/proc 还能查）
- 安全审计的基础（auditd 的 /proc 规则依然有效）

0.7 第 6 章：容器时代的 /proc 攻防——从 runc 到 K8s

如果说 Linux 内核社区在「绕过」/proc，那么容器社区就是在「重新发明」/proc。

容器本质上是一种**虚拟化的极端形式**——它把 Linux 内核的进程、网络、文件系统 namespace 拆分成多份，让每个容器以为自己是独占的。在这场拆分中，/proc 是最复杂的部分。

0.7.1 容器里 /proc 的特殊地位

在一个容器里，/proc 是**唯一能感知「自己在容器内」**的接口：

- /proc/self/cgroup —当前进程所在的 cgroup 路径
- /proc/[pid]/ns/* —namespace 信息
- /proc/[pid]/status —PID namespace 信息
- /proc/[pid]/mountinfo —mount namespace 信息

但容器里 /proc 也是**最大的攻击面**：

- 如果 /proc 没正确隔离，容器内能看到宿主 PID
- 如果 /proc/sys 没正确限制，容器内能修改内核参数
- 如果 /proc/[pid]/mem 可读，容器内能读其他容器进程内存

0.7.2 容器运行时（runc / containerd / CRI-O）的 /proc 处理

0.7.2.1 runc 默认的 /proc 配置

```
// runc/libcontainer/specs/config.go（简化）
defaultProcMask = []string{
    "/proc/asound",
    "/proc/acpi",
    "/proc/kcore",
    "/proc/keys",
```

```
    "/proc/latency_stats",
    "/proc/timer_list",
    "/proc/timer_stats",
    "/proc/sched_debug",
    "/proc/scsi",
    "/proc/sysrq-trigger",
}
```

容器启动时，runc 会：

1. 把宿主 /proc bind mount 进容器
2. 用 maskedPaths 把上述路径替换为 /dev/null（屏蔽）
3. 用 readOnlyPaths 把更多路径设为只读
4. 用 mount propagation 限制容器内 mount 不会泄露到宿主

0.7.2.2 maskedPaths vs readOnlyPaths

- **maskedPaths**：路径被替换为 /dev/null，任何访问都返回空
- **readOnlyPaths**：路径可读但不可写

```
{
    "ociVersion": "1.0.0",
    "process": {
        "capabilities": {
            "bounding": [...],
            "effective": [...],
            "inheritable": [],
            "permitted": [...]
        }
    },
    "rootfs": {
        "path": "rootfs"
    },
    "mounts": [
        {
            "destination": "/proc",
            "type": "proc",
            "source": "proc"
        }
    ],
    "linux": {
        "maskedPaths": [
            "/proc/asound",
            "/proc/acpi",
            ...
        ],
        "readOnlyPaths": [
            "/proc/bus",
            "/proc/fs",
            "/proc/irq",
            "/proc/sys",
            "/proc/sysrq-trigger"
        ]
    }
}
```

0.7.2.3 CVE-2025-52881：runc 的 maskedPath 绕过

2025 年披露的 CVE-2025-52881 揭示了 maskedPath 的一个**严重绕过**：

- 攻击者在容器内创建符号链接 /dev/null -> /proc/sys/kernel/core_pattern
- maskedPath 默认的实现是「用 /dev/null 覆盖原路径」，但**没考虑符号链接攻击**
- 结果：容器内对 /proc/sys/kernel/core_pattern 的写入实际写到了宿主内核参数

- 后果：**容器逃逸 → 宿主机 root**

修复：runc 1.2.8 之后，maskedPath 用「两步 mount」实现：

```
# 第一步：bind mount 原路径
mount --bind /proc/sys/kernel/core_pattern /proc/sys/kernel/core_pattern

# 第二步：remount 为只读
mount -o remount,ro,bind /proc/sys/kernel/core_pattern
```

两步 mount 的关键：bind mount 之后再 remount，会先固化原路径，再做 remount。符号链接攻击在第一步 bind 时被「固化」了。

0.7.2.4 检查你的 runc 版本

```
runc --version
# 应该是 1.2.8 或更高
```

低于 1.2.8 的版本都受 CVE-2025-52881 影响。

0.7.3 容器内的 /proc 攻击案例

0.7.3.1 案例 1：读宿主进程的环境变量

前提：runc 没启用 hidepid，或者容器共享了宿主的 /proc。

```
# 在容器内
cat /proc/1/environ
# 输出：PATH=/usr/local/sbin:...container=docker...
# → 暴露了宿主的 PATH 等环境
```

如果宿主 PID 1 是 systemd，攻击者可能从环境变量里拿到：

- 容器管理器的配置
- 服务发现的密钥
- 数据库连接字符串

0.7.3.2 案例 2：写 /proc/sys/kernel/core_pattern

前提：runc 老版本（< 1.2.8），或 readOnlyPaths 没覆盖 /proc/sys。

```
# 在容器内
echo '|/tmp/x' > /proc/sys/kernel/core_pattern
# 让某个进程 crash
kill -SIGSEGV $$
# → 内核执行 /tmp/x，以 root 身份
```

这就是 CVE-2025-52881 的攻击路径。

0.7.3.3 案例 3：通过 /proc/[pid]/ns/pid 越界

前提：容器共享宿主的 PID namespace。

```
# 在容器内
ls -la /proc/1/ns/pid
# 输出：lrwxrwxrwx ... pid -> pid:[4026531836]

# 如果宿主 PID 1 是 systemd，可以：
nsenter -t 1 -p -m -i -u -n -- /bin/bash
# → 进入宿主的 namespace，拿到宿主 shell
```

0.7.3.4 案例 4：/proc/self/root 越界（最隐蔽）

前提：容器内的 rootfs 是 chroot 而不是真正的 mount namespace 隔离。

```
# 在容器内
ls -la /proc/self/root
# 输出: lrwxrwxrwx ... root -> /

# 通过修改 /proc/self/root 的目标，可以越界
# （攻击复杂度高，但完全可行）
```

0.7.4 容器内的 /proc 防御

0.7.4.1 K8s PodSpec 的安全配置

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:
    # 1. 容器必须以非 root 运行
    runAsNonRoot: true
    runAsUser: 1000

    # 2. 限制 capability
    capabilities:
      drop:
        - ALL

    # 3. 不允许特权模式
    allowPrivilegeEscalation: false

    # 4. seccomp profile（必填）
    seccompProfile:
      type: RuntimeDefault

    # 5. AppArmor profile
    appArmorProfile:
      type: RuntimeDefault

  containers:
  - name: app
    image: myapp:1.0
    securityContext:
      # 1. 只读 rootfs
      readOnlyRootFilesystem: true

      # 2. 禁止写入 /proc/sys
      procMount: "Default"

      # 3. SELinux options
      seLinuxOptions:
        level: "s0:c123,c456"
```

0.7.4.2 procMount: "Default" vs procMount: "Unmasked"

K8s 1.12+ 引入 procMount 字段：

- "Default"：使用 runc 的 maskedPaths / readOnlyPaths 默认配置
- "Unmasked"：**绕过** maskedPaths，容器内可以访问 /proc/asound 等

```
securityContext:
  procMount: "Default"  # 永远用 Default
```

Unmasked 几乎永远不应该用。它存在的唯一原因是某些特殊应用（比如实时音频）需要访问 /proc/asound，但这应该用 device plugin 而不是 Unmasked。

0.7.4.3 Podman / Docker 的 --security-opt

非 K8s 场景（Podman、Docker）用 --security-opt：

```
podman run -d \
  --name myapp \
  --security-opt=no-new-privileges \
  --security-opt seccomp=/path/to/seccomp.json \
  --security-opt apparmor=myapp-restricted \
  --security-opt procfs=Default \
  --read-only \
  --tmpfs /tmp \
  myapp:1.0
```

0.7.4.4 no-new-privileges flag

这是**最重要**的容器安全 flag。它禁止容器内进程通过 setuid binary 提权：

```
# 不加 no-new-privileges:
# 容器内有 su / sudo / passwd，攻击者可以用它们提权
# 即使你把 capability 都 drop 了

# 加 no-new-privileges:
# 攻击者无法用 su / sudo 提权
# → 必须寻找其他路径（更难）
```

0.7.5 K8s 时代的 /proc 安全策略自动化

0.7.5.1 Kyverno / OPA Gatekeeper 策略

用策略引擎强制所有 Pod 配置安全选项：

```
# kyverno policy
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-proc-security
spec:
  validationFailureAction: Enforce
  rules:
  - name: check-procMount
    match:
      any:
      - resources:
          kinds:
          - Pod
    validate:
      message: "procMount must be Default"
      pattern:
        spec:
          containers:
          - securityContext:
              procMount: "Default"

  - name: check-no-new-privileges
    match:
      any:
      - resources:
          kinds:
          - Pod
    validate:
      message: "allowPrivilegeEscalation must be false"
      pattern:
        spec:
```

```
        containers:
        - securityContext:
            allowPrivilegeEscalation: false
```

0.7.5.2 Kube-bench / Kube-hunter 扫描

```
# 扫描集群是否符合 CIS Benchmark
kube-bench run --targets node

# 扫描集群的攻击面
kube-hunter --remote some-cluster.example.com
```

这些工具能识别：

- 容器没设 runAsNonRoot
- 容器没 drop capabilities
- 容器用了 procMount: Unmasked
- 容器没用 no-new-privileges

0.7.5.3 Falco 的容器规则集

Falco 默认规则集专门有容器逃逸检测：

```
# Falco default rules (subset)
- rule: Container Drift Detected
  desc: New container started with unexpected attributes
  condition: >
    container_started
    and not container.image.repository in (allowed_images)
  output: ...
  priority: WARNING

- rule: Sensitive Mount from Host
  desc: Container has /proc/sys mounted from host
  condition: >
    container_mount and container_mount.source.path in (/proc/sys, /proc/sysrq-trigger)
  output: ...
  priority: CRITICAL
```

0.7.5.4 Tetragon 的运行时防御

Tetragon 能 hook 容器内的进程行为：

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: block-proc-write
spec:
  kprobes:
  - call: "vfs_write"
    syscall: false
    args:
    - index: 0
      type: "nop"
    selectors:
    - matchArgs:
      - index: 0
        operator: "Prefix"
        values:
        - "/proc/sys/kernel/core_pattern"
      matchNamespaces:
      - namespace: "default"
        operator: "NotIn"
```

```
    matchActions:
    - action: Sigkill
```

这个 TracingPolicy 的含义：**如果 default namespace 外的进程写 /proc/sys/kernel/core_pattern，直接 Sigkill**。这就是 CVE-2025-52881 的运行时防御。

0.7.6 镜像供应链的 /proc 安全

0.7.6.1 镜像里的恶意 /proc 写入

有些恶意镜像会在 entrypoint 脚本里做敏感操作：

```
# 恶意 entrypoint 示例
#!/bin/bash
# 检查 /proc/sys/kernel/core_pattern 是否可写
if [ -w /proc/sys/kernel/core_pattern ]; then
  # 把 core dump 处理程序改成反弹 shell
  echo '|/tmp/backdoor' > /proc/sys/kernel/core_pattern
fi

# 启动应用
exec /usr/bin/myapp
```

检测：用 dive 工具看镜像的 layer：

```
dive myapp:1.0
# 看 entrypoint.sh 的内容
# 看是否有可疑操作
```

0.7.6.2 Distroless 镜像

Distroless 镜像（Google 出品）只包含应用 + 必要的 runtime，**不包含 shell / package manager**：

```
FROM gcr.io/distroless/base-debian12
COPY --from=builder /app/myapp /myapp
ENTRYPOINT ["/myapp"]
```

好处：- 攻击者拿到容器 shell 后，能执行的操作极少 - 没有 shell 就无法 echo > /proc/sys/... - 没有 curl / wget 就无法下载 payload

0.7.6.3 镜像签名与 SBOM

```
# 用 cosign 签名镜像
cosign sign --key cosign.key myapp:1.0

# 验证签名
cosign verify --key cosign.pub myapp:1.0

# 生成 SBOM
syft myapp:1.0 -o spdx-json > sbom.spdx.json
```

签名确保镜像在供应链上没被篡改。SBOM（Software Bill of Materials）让你知道镜像里到底有什么。

0.7.7 容器沙箱：/proc 隔离的新方向

容器社区也在探索**比 runc 更强的沙箱**：

0.7.7.1 gVisor

Google 的 gVisor 在**用户态实现了一个内核**，拦截容器内所有 syscall：

```
App → syscalls → gVisor Sentry → host kernel
                      ↓
                /proc (gVisor 实现)
```

gVisor 里的 /proc 是**虚拟的**——它拦截容器的 /proc 访问，自己构造响应。结果是：**容器内完全看不到宿主 /proc**。

代价：性能损失 10-30%。但对于多租户场景，这是值得的。

0.7.7.2 Kata Containers

Kata 用**轻量虚拟机**跑容器：

```
Container → runc (sandbox) → QEMU / Firecracker → host kernel
                                        ↓
                                  /proc (own kernel)
```

Kata 的 /proc 是**虚拟机自己的**——和宿主完全隔离。性能损失 5-10%，但隔离性接近 VM。

0.7.7.3 Firecracker

AWS 的 Firecracker 把 VM 启动时间压到 125ms，内存开销压到 5MB，让「VM 级隔离」的成本接近容器。

未来方向：**容器 + VM 的边界会越来越模糊**。

0.7.8 容器内 /proc 的可观测性

容器内的 /proc 有**特殊的可观测性挑战**：

- /proc/[pid]/io —容器内进程的 I/O 统计，但只能看到容器内的 PID
- /proc/net/tcp —容器看到的网络，是容器 namespace 内的网络

0.7.8.1 cAdvisor

cAdvisor 自动发现容器内的 /proc 资源使用：

```yaml
# Prometheus 抓取 cAdvisor
scrape_configs:
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
```

cAdvisor 指标包括：

- container_cpu_usage_seconds_total
- container_memory_usage_bytes
- container_network_*_bytes_total

0.7.8.2 eBPF 跨容器观测

eBPF 的优势在容器场景更明显——**一次 hook，所有容器可见**：

```
// 监控所有容器内的 /proc 写入
SEC("kprobe/vfs_write")
int kprobe__vfs_write(struct pt_regs *ctx, struct file *file,
                      const char __user *buf, size_t count, loff_t *pos) {
    char filename[256] = {};
    bpf_probe_read_str(filename, sizeof(filename),
                       file->f_path.dentry->d_name.name);

    char parent[256] = {};
    bpf_probe_read_str(parent, sizeof(parent),
                       file->f_path.dentry->d_parent->d_name.name);

    // 检查是否是 /proc/sys/kernel/core_pattern
    if (strstr(filename, "core_pattern") &&
        strstr(parent, "kernel")) {
        u32 pid = bpf_get_current_pid_tgid() >> 32;
        // 把 container id 也带上
        bpf_printk("pid=%d write core_pattern\\n", pid);
```

```
    }
    return 0;
}
```

0.7.9 小结

容器时代的 /proc 安全是**多层防御**：

1. **镜像层**：Distroless、签名、SBOM
2. **配置层**：PodSpec 安全选项、procMount=Default
3. **运行时层**：runc 最新版、Tetragon / Falco 运行时检测
4. **沙箱层**：gVisor / Kata Containers 的更强隔离
5. **可观测层**：eBPF 跨容器观测

没有银弹。每层都有自己的代价（性能、复杂度、运维成本）。安全团队要做的，是**根据业务威胁模型选择合适的层组合**。

下一章我们收尾——讲「/proc 在 SRE / 安全工具链中的位置」。

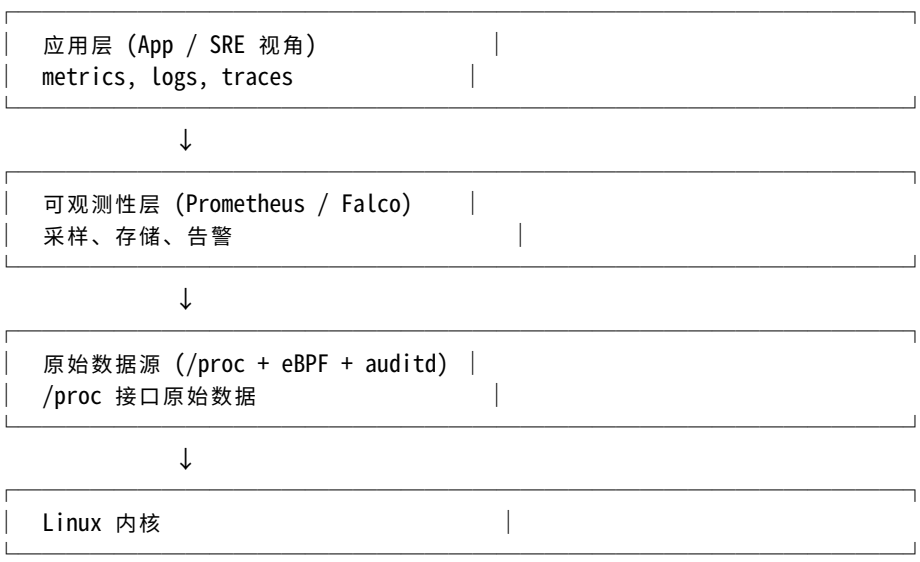
0.8 第 7 章：从 /proc 到 SRE / 安全工具链的整合——最后一公里

写到这里，我们已经走完了 /proc 这座山的全貌。但一座山的「山顶」和「山脚」之间，还有一个被很多人忽略的「最后一公里」——怎么把 /proc 的能力**整合进日常的 SRE / 安全工具链**。

这一章不讲新知识，只讲**怎么把前面 6 章的内容「用起来」**。

0.8.1 定位：/proc 在工具链中的位置

在 SRE / 安全的工具链里，/proc 扮演着**底层数据源 + 应急排查接口**的双重角色：



/proc 在这个金字塔里是**基座**。其他工具都「站在它上面」。

这意味着什么：

1. 任何监控工具、IDS、审计系统**都依赖 /proc 提供原始数据**
2. /proc 的接口变更会**波及整个可观测性生态**（这是为什么 /proc 不能轻易改）
3. 当其他工具都失效时，/proc 是**最后一道应急排查接口**

0.8.2 与上游工具的集成

0.8.2.1 node_exporter：最基础的集成

node_exporter 把 /proc 接口转化为 Prometheus 指标，是所有 Linux 服务器可观测性的事实标准。

深度集成套路：

```
# /etc/default/prometheus-node-exporter
# 关键启动参数
ARGS="\
  --collector.systemd \
  --collector.processes \
  --collector.filesystem.mount-points-exclude='^/(sys|proc|dev)($|/)' \
  --collector.netclass.ignored-devices='^(veth.*|docker.*)$' \
  --web.max-requests=50 \
"
```

补充采集器：

```
# 文本采集器：用 --collector.textfile.directory
# /etc/prometheus/textfile_collector/
# 丢 .prom 文件进去，会被采集

# 示例：脚本生成一个文本采集器输出
cat > /etc/cron.daily/proc_textfile_collector << 'EOF'
#!/bin/bash
OUTPUT="/etc/prometheus/textfile_collector/proc_extra.prom"
echo "# HELP proc_world_readable_count /proc 中其他用户可读文件数" > "$OUTPUT"
echo "# TYPE proc_world_readable_count gauge" >> "$OUTPUT"
COUNT=$(find /proc -maxdepth 3 -perm -o+r -type f 2>/dev/null | wc -l)
echo "proc_world_readable_count $COUNT" >> "$OUTPUT"
EOF
chmod +x /etc/cron.daily/proc_textfile_collector
```

0.8.2.2 Prometheus：告警规则的合理分组

```
# /etc/prometheus/rules/proc.yml
groups:
- name: proc_baseline
  interval: 30s
  rules:
  # 总体进程数（异常 fork bomb / 进程泄漏）
  - alert: ProcessCountAnomaly
    expr: node_processes_total > (avg_over_time(node_processes_total[7d]) * 1.5)
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: " 进程数异常（当前 {{ $value }}，7 天均值 × 1.5）"

  # 状态为空的进程（隐藏进程的可疑迹象）
  - alert: HiddenProcessSuspect
    expr: proc_comm_empty_total > 0
    for: 1m
    labels:
      severity: critical

- name: proc_network
  interval: 30s
  rules:
  # TCP 连接数激增
  - alert: TCPEstablishedSpike
    expr: proc_tcp_state_established > (avg_over_time(proc_tcp_state_established[7d]) * 2)
```

```
    for: 5m
    labels:
      severity: warning

- name: proc_io
  interval: 30s
  rules:
  # I/O 高峰
  - alert: IOHighByProcess
    expr: topk(5, rate(proc_io_write_bytes_total[5m])) > 1e8
    for: 10m
    labels:
      severity: info
```

0.8.2.3 Grafana：/proc 数据的可视化

核心 dashboard 模板：

```
{
  "title": "/proc 可观测性概览",
  "panels": [
    {
      "title": " 进程状态分布",
      "type": "stat",
      "targets": [
        {
          "expr": "sum(node_processes_state) by (state)",
          "legendFormat": "{{state}}"
        }
      ]
    },
    {
      "title": "TCP 连接状态",
      "type": "timeseries",
      "targets": [
        {
          "expr": "proc_tcp_state_established",
          "legendFormat": "ESTABLISHED"
        },
        {
          "expr": "proc_tcp_state_listen",
          "legendFormat": "LISTEN"
        },
        {
          "expr": "proc_tcp_state_time_wait",
          "legendFormat": "TIME_WAIT"
        }
      ]
    },
    {
      "title": "I/O 热点进程 Top 10",
      "type": "table",
      "targets": [
        {
          "expr": "topk(10, sum by (comm) (rate(proc_io_write_bytes_total[5m])))",
          "format": "table",
          "instant": true
        }
      ]
    },
    {
      "title": "Fd 异常进程",
```

```
      "type": "table",
      "targets": [
        {
          "expr": "proc_fd_count_total > 200",
          "format": "table"
        }
      ]
    },
    {
      "title": "/proc 访问审计事件",
      "type": "timeseries",
      "targets": [
        {
          "expr": "sum by (key) (rate(audit_proc_events_total[5m]))",
          "legendFormat": "{{key}}"
        }
      ]
    }
  ]
}
```

可以直接导入：把上面这个 JSON 存为 `proc-overview-dashboard.json`，在 Grafana 里 `Import`。

0.8.3 与下游工具的集成

0.8.3.1 SIEM：把 /proc 事件导入 Splunk / ELK

```
# 用 filebeat 把 auditd 日志送到 ELK
# /etc/filebeat/filebeat.yml
filebeat.inputs:
- type: log
  paths:
    - /var/log/audit/audit.log
  fields:
    source: auditd
  processors:
  - dissect:
      tokenizer: '%{}type=%{} msg=audit(%{ts}): %{event}'
      field: "message"

output.elasticsearch:
  hosts: ["elk.internal:9200"]
  index: "auditd-%{+yyyy.MM.dd}"
```

0.8.3.2 SOAR：把告警变成 playbook

```
# Splunk SOAR playbook 示例
def playbook_proc_environ_read(event):
    if event['severity'] == 'critical':
        # 1. 自动隔离进程
        kill_cmd = f"kill -9 {event['pid']}"
        exec_on_target(event['host'], kill_cmd)

        # 2. 抓进程内存 dump（取证）
        gcore_cmd = f"gcore {event['pid']}"
        exec_on_target(event['host'], gcore_cmd)

        # 3. 通知 oncall
        notify_oncall(event)

        # 4. 创建 Jira 工单
        create_jira_ticket(
            title=f"Critical /proc 访问: {event['description']}",
```

```
            severity="P0",
        )
```

0.8.3.3 ITSM：把告警关联到变更

```yaml
# ServiceNow 集成示例
- alert: ProcEnvironRead
  webhook:
    url: https://servicenow.internal/api/incident
    payload:
      short_description: " 检测到 /proc/[pid]/environ 异常读取"
      severity: 2
      assignment_group: "Security Team"
      category: "Security"
```

0.8.4 /proc 应急响应剧本（IR Playbook）

当 /proc 相关的告警触发时，oncall 应该做什么？我们给出**标准化的应急响应剧本**。

0.8.4.1 剧本 1：检测到异常 /proc/[pid]/environ 读取

触发条件：proc-environ-read 告警 + 非白名单进程

响应步骤：

```
# T+0: 收到告警（30 秒内）
# Step 1: 看告警详情
# - 哪个 PID?
# - 哪个用户?
# - 读的目标 PID 是哪个?

# T+5 分钟
# Step 2: 隔离可疑进程
kill -STOP <pid>     # 暂停，但不杀
# → 让取证脚本能完整 dump

# T+10 分钟
# Step 3: 取证 dump
mkdir -p /tmp/ir-<pid>
cd /tmp/ir-<pid>
gcore <pid>            # 内存 dump
cat /proc/<pid>/cmdline | tr '\0' ' ' > cmdline.txt
cat /proc/<pid>/status > status.txt
cat /proc/<pid>/maps > maps.txt
ls -la /proc/<pid>/fd/ > fd.txt
cat /proc/<pid>/io > io.txt
cat /proc/<pid>/status | grep -E '^(Name|Pid|PPid|State|Uid|Gid)' > basic.txt

# T+15 分钟
# Step 4: 杀进程 + 网络隔离
kill -9 <pid>
iptables -A OUTPUT -m owner --uid-owner <uid> -j DROP

# T+20 分钟
# Step 5: 影响范围评估
# - 这台机器上的其他进程有类似行为吗?
ausearch -k proc-environ-read --interpret | tail -100
# - 这台机器最近 24h 有什么可疑行为?
ausearch --start yesterday --end now --interpret | grep -E "core_pattern|modules"

# T+30 分钟
# Step 6: 通知 + 工单
# - 在 wecom 通知 oncall 群
```

```
# - 创建 P1 Jira 工单
# - 邮件发给安全主管
```

0.8.4.2 剧本 2：检测到 /proc/sys/kernel/core_pattern 写入

触发条件：proc-core-pattern-write 告警

响应步骤：

```
# T+0: 收到告警
# 这是 P0 级别——可能是容器逃逸

# T+1 分钟
# Step 1: 立即恢复 core_pattern 为 default
echo 'core' > /proc/sys/kernel/core_pattern

# T+2 分钟
# Step 2: 隔离容器
# 如果是 K8s pod: cordon node + delete pod
kubectl cordon <node>
kubectl delete pod --all-namespaces --field-selector spec.nodeName=<node>

# T+5 分钟
# Step 3: 排查入侵源
# - 是哪个进程写的?
ausearch -k proc-core-pattern-write --interpret
# - 它的可执行文件路径是什么?
ls -la /proc/<pid>/exe
sha256sum $(readlink /proc/<pid>/exe)
# - 它从哪里来?
cat /proc/<pid>/cmdline
cat /proc/<pid>/cgroup

# T+15 分钟
# Step 4: 影响范围评估
# - 其他机器有类似 core_pattern 写入吗?
grep -r "core_pattern" /var/log/audit/ 2>/dev/null
# - 当前 runc 版本是多少?
runc --version
# 如果 < 1.2.8, 全集群都要升级

# T+30 分钟
# Step 5: 升级 runc
apt-get update && apt-get install --only-upgrade runc
# 或者在 K8s 里升级 containerd / CRI-O
```

0.8.4.3 剧本 3：检测到异常 fd 数量

触发条件：proc-fd-anomaly 告警（单进程 fd $>$ 200）

响应步骤：

```
# Step 1: 看进程信息
ps -p <pid> -o pid,user,comm,cmd
cat /proc/<pid>/status

# Step 2: 看 fd 列表
ls -la /proc/<pid>/fd/ | head -50

# Step 3: 看 fd 类型分布
ls -la /proc/<pid>/fd/ | awk '{print $NF}' | \
  sed -E 's/\[(.*)\]/\1/' | sort | uniq -c | sort -rn

# 典型输出:
```

```
#   50 socket  ← 50 个 socket（异常）
#    5 pipe
#    3 file

# Step 4: socket fd 反查连接
for fd in /proc/<pid>/fd/*; do
  if [[ "$(readlink $fd)" =~ socket:\[([0-9]+)\] ]]; then
    inode=${BASH_REMATCH[1]}
    # 在 /proc/net/tcp 找这个 inode
    grep " $inode " /proc/net/tcp /proc/net/tcp6 2>/dev/null
  fi
done

# Step 5: 隔离
kill -STOP <pid>
# 进一步分析
```

0.8.5 /proc 知识的团队传播

/proc 的能力**不是只靠工具链就够的**。它需要**人的能力**作为支撑。

0.8.5.1 「/proc 红宝书」——团队内训材料

很多团队的 /proc 知识分散在老员工脑子里。**这不行**。需要把 /proc 的关键知识写成**红宝书**，定期内训。

红宝书的最小集合：

1. /proc/[pid]/status 怎么读？（state、PPid、Uid、Gid、Threads）
2. /proc/[pid]/maps 怎么看？（堆、栈、lib、anon、file-backed）
3. /proc/net/tcp 怎么解析？（hex IP、hex port、状态机）
4. /proc/sys/kernel/core_pattern 的含义？
5. hidepid 怎么用？
6. 怎么从 fd 反查 socket 到 PID？

0.8.5.2 「/proc 实战演练」

每季度一次的**红蓝对抗**，专门测试 /proc 的应急响应能力：

```
# 演练剧本示例


# 版本信息

> 本页为电子书的自动生成元数据，会出现在 PDF / ePub / HTML 三个格式的最后一页。

| 字段 | 值 |
|------|----|
| 当前版本 | v1.0.0 |
| 下一待发布版本 | v1.1.0 |
| 发布日期 | 2026 年 8 月 1 日 |
| 仓库 | https://github.com/LeisureLinux/ebook-procfs-hardening |
| 在线阅读 | https://leisurelinux.github.io/ebook-procfs-hardening/ |
| 许可 | MIT License — © 2026 LeisureLinux |

## 本次版本修订记录（v1.0.0）

- feat: SEO/GEO overhaul — llms.txt, audit script, structured data, sitemap
- docs: cross-link to ssh-hardening sister ebook

## 历史版本

| 版本 | 日期 | 主要变更 |
```

```
|------|------|----------|
| v1.0.0 | 2026-08-01 | feat: SEO/GEO overhaul — llms.txt, audit script, structured data, sitemap |
| v0.3 | 2026-07-23 | fix(ebook): add hyperlinks to appendix C (CVE/books/tools/man pages) |
| v0.2 | 2026-07-23 | fix(ebook): restore line continuation after --number-sections removal |
| v0.1 | 2026-07-23 | Initial commit: 7 chapters + 3 appendices for /proc defense in depth ebook |

---

> 本书使用 [Pandoc](https://pandoc.org/) + [XeLaTeX](https://tug.org/xetex/) 构建，
> 字体采用 [Noto Serif CJK SC](https://github.com/notofonts/noto-cjk)，
> 通过 GitHub Actions 自动构建并发布到 GitHub Pages。

## 附录 A: Linux 6.x `/proc` 接口速查表

> 本表整理 Linux 6.x 内核（以 6.6 / 6.10 为基准）所有与安全、可观测性相关的 `/proc` 接口。完整列表请参考 `man 5 proc`。

### A.1 系统级 `/proc` 接口

| 路径 | 主要内容 | 安全影响 | 可读性 |
|---|---|---|---|
| `/proc/cpuinfo` | CPU 信息 | 泄露 CPU 型号（决定可利用 CVE） | 全员可读 |
| `/proc/meminfo` | 内存统计 | 泄露系统内存（DoS 门槛判断） | 全员可读 |
| `/proc/cmdline` | 启动参数 | ** 泄露内核参数、调试选项 ** | 全员可读 |
| `/proc/mounts` | 当前挂载 | 泄露敏感路径 | 全员可读 |
| `/proc/filesystems` | 支持的文件系统 | 信息泄露 | 全员可读 |
| `/proc/swaps` | swap 状态 | 信息泄露 | 全员可读 |
| `/proc/version` | 内核版本 | ** 决定可利用 CVE** | 全员可读 |
| `/proc/uptime` | 启动时长 | 信息泄露 | 全员可读 |
| `/proc/loadavg` | 平均负载 | 信息泄露 | 全员可读 |
| `/proc/sys/kernel/core_pattern` | core dump 处理 | ** 可写 → 提权 ** | 仅 root 可写 |
| `/proc/sys/kernel/modules_disabled` | 模块加载开关 | 写到 1 永久禁用模块 | 仅 root 可写 |
| `/proc/sys/kernel/yama/ptrace_scope` | ptrace 限制 | ** 决定进程间调试权限 ** | 全员可读 |
| `/proc/sys/kernel/randomize_va_space` | ASLR | 写到 0 关闭 ASLR | 仅 root 可写 |
| `/proc/sys/kernel/kptr_restrict` | 内核符号隐藏 | ** 写到 1+ 隐藏符号 ** | 全员可读 |
| `/proc/sys/kernel/dmesg_restrict` | dmesg 限制 | ** 写到 1 限制 dmesg** | 全员可读 |
| `/proc/sys/kernel/unprivileged_bpf_disabled` | BPF 限制 | ** 写到 2 永久禁用非特权 BPF** | 全员可读 |
| `/proc/sys/kernel/usermodehelper/binfmt_misc` | usermodehelper | CVE-2025-31133 缓解 | 全员可读 |
| `/proc/sys/fs/protected_hardlinks` | hardlink 保护 | 写到 1 限制 hardlink | 全员可读 |
| `/proc/sys/fs/protected_symlinks` | symlink 保护 | 写到 1 限制 symlink | 全员可读 |
| `/proc/sys/fs/protected_fifos` | FIFO 保护 | 写到 1+ 限制 FIFO | 全员可读 |
| `/proc/sys/fs/protected_regular` | 常规文件保护 | 写到 1+ 限制 | 全员可读 |
| `/proc/sys/net/ipv4/conf/all/route_filter` | 反 spoof | 写到 1 限制 | 全员可读 |
| `/proc/sys/net/ipv4/icmp_echo_ignore_all` | ping 限制 | 写到 1 忽略 ping | 全员可读 |
| `/proc/sys/vm/mmap_min_addr` | mmap 最低地址 | ** 写到 65536+ 防止空指针 ** | 全员可读 |
| `/proc/sys/kernel/sysrq` | SysRq 限制 | 写到 0 禁用 SysRq | 仅 root 可写 |
| `/proc/sys/kernel/kexec_load_disabled` | kexec 限制 | 写到 1 禁用 kexec | 仅 root 可写 |
| `/proc/sys/kernel/perf_event_paranoid` | perf 限制 | 写到 2+ 限制 perf | 全员可读 |

### A.2 进程级 `/proc/[pid]/*` 接口

| 路径 | 主要内容 | 安全影响 | 默认权限 |
|---|---|---|---|
| `/proc/[pid]/status` | 进程状态（state、Pid、PPid、Uid、Gid、Threads） | ** 定位隐藏进程 ** | 644 |
| `/proc/[pid]/cmdline` | 命令行 | ** 全路径泄露 ** | 440 |
| `/proc/[pid]/environ` | 环境变量 | ** 明文凭证泄露 ** | 400 |
| `/proc/[pid]/cwd` | 当前工作目录 | 符号链接 | 777 (symlink) |
| `/proc/[pid]/exe` | 可执行文件 | 符号链接 | 777 (symlink) |
| `/proc/[pid]/maps` | 内存映射 | ** 内存布局发现 ** | 444 |
| `/proc/[pid]/mem` | 进程内存 | ** 可读 → 凭证窃取 ** | 440 (受 ptrace 限制) |
| `/proc/[pid]/io` | I/O 统计 | I/O 发现 | 444 |
| `/proc/[pid]/stat` | 进程状态机 | 进程发现 | 444 |
| `/proc/[pid]/statm` | 内存使用 | 信息泄露 | 444 |
```

`/proc/[pid]/fd/`	文件描述符	**socket fd → C2 通道反查 **	500
`/proc/[pid]/fdinfo/[fd]`	fd 信息	socket inode 反查	500
`/proc/[pid]/task/[tid]/`	线程	线程发现	同上
`/proc/[pid]/ns/`	namespace	** 越界检测 **	444
`/proc/[pid]/mountinfo`	mount namespace	隔离发现	444
`/proc/[pid]/cgroup`	cgroup 路径	** 精确定位容器 **	444
`/proc/[pid]/attr/current`	SELinux context	信息泄露	444
`/proc/[pid]/wchan`	等待通道	内核信息泄露（**kptr_restrict 控 **）	444
`/proc/[pid]/stack`	内核调用栈	** 受 kptr_restrict 控 **	444
`/proc/[pid]/smaps`	详细内存映射	** 受 ptrace 限制 **	440
`/proc/[pid]/smaps_rollup`	smaps 汇总	受 ptrace 限制	440
`/proc/[pid]/pagemap`	物理页映射	** 受 CAP_SYS_ADMIN 限制 **	400
`/proc/[pid]/oom_score`	OOM 评分	信息泄露	444
`/proc/[pid]/oom_score_adj`	OOM 调整	可写（自己有写权限）	644
`/proc/[pid]/comm`	进程名	** 可被攻击者修改 **（root）	644
`/proc/[pid]/limits`	资源限制	信息泄露	444
`/proc/[pid]/sched`	调度信息	信息泄露	444
`/proc/[pid]/schedstat`	调度统计	信息泄露	444
`/proc/[pid]/timerslack_ns`	timer slack	可写（自己有写权限）	644
`/proc/[pid]/setgroups`	setgroups 标志	受 CAP_SETGID 限制	644
`/proc/[pid]/uid_map`	UID 映射	user namespace 边界	644
`/proc/[pid]/gid_map`	GID 映射	user namespace 边界	644
`/proc/[pid]/loginuid`	登录 UID	审计追踪	444
`/proc/[pid]/sessionid`	会话 ID	审计追踪	444
`/proc/[pid]/timers`	POSIX timer	信息泄露	444

A.3 网络级 `/proc/net/*` 接口

路径	主要内容	安全影响	默认权限
`/proc/net/tcp`	TCP 连接（v4）	** 连接发现、攻击路径 **	444
`/proc/net/tcp6`	TCP 连接（v6）	同上	444
`/proc/net/udp`	UDP socket	信息泄露	444
`/proc/net/udp6`	UDP socket（v6）	信息泄露	444
`/proc/net/unix`	Unix domain socket	信息泄露	444
`/proc/net/raw`	raw socket	信息泄露	444
`/proc/net/arp`	ARP 表	** 内网拓扑 **	444
`/proc/net/route`	路由表	信息泄露	444
`/proc/net/ip_conntrack`	conntrack 表	** 连接跟踪 **	444
`/proc/net/dev`	设备统计	信息泄露	444
`/proc/net/snmp`	SNMP 统计	信息泄露	444
`/proc/net/netstat`	详细统计	信息泄露	444
`/proc/net/sockstat`	socket 总数	信息泄露	444
`/proc/net/rpc`	RPC 状态（NFS）	**NFS 暴露面 **	444
`/proc/net/fib_trie`	FIB trie	路由内部信息	444
`/proc/net/if_inet6`	IPv6 接口	信息泄露	444
`/proc/net/igmp`	IGMP	信息泄露	444
`/proc/net/ip_mr_cache`	IP multicast cache	信息泄露	444
`/proc/net/ip_mr_vifs`	IP multicast VIF	信息泄露	444
`/proc/net/psched`	packet scheduler	信息泄露	444
`/proc/net/ptype`	packet types	信息泄露	444
`/proc/net/softnet_perf`	softnet 性能	信息泄露	444
`/proc/net/stat/conntrack`	conntrack 统计	信息泄露	444
`/proc/net/stat/ipsec`	IPsec 统计	信息泄露	444

A.4 容器相关 `/proc` 接口

路径	主要内容	安全影响
`/proc/self/cgroup`	当前 cgroup	** 容器定位 **
`/proc/[pid]/cgroup`	目标 PID 的 cgroup	跨容器追踪
`/proc/[pid]/ns/pid`	PID namespace	** 越界检测 **

`/proc/[pid]/ns/mnt`	mount namespace	越界检测
`/proc/[pid]/ns/net`	network namespace	越界检测
`/proc/[pid]/ns/ipc`	IPC namespace	越界检测
`/proc/[pid]/ns/user`	user namespace	越界检测
`/proc/[pid]/ns/uts`	UTS namespace	越界检测
`/proc/[pid]/ns/cgroup`	cgroup namespace	越界检测
`/proc/[pid]/ns/time`	time namespace (6.7+)	时间越界
`/proc/[pid]/mountinfo`	mount 详情	越界检测
`/proc/[pid]/mounts`	mount 简版	越界检测

A.5 系统控制 `/proc/sys/` 子目录

子目录	内容
`/proc/sys/kernel/`	内核参数（core_pattern、modules_disabled 等）
`/proc/sys/vm/`	虚拟内存参数（mmap_min_addr、overcommit_memory 等）
`/proc/sys/fs/`	文件系统参数（protected_*、file-max 等）
`/proc/sys/net/`	网络参数（ipv4、ipv6 子树）
`/proc/sys/crypto/`	加密子系统（很少用）
`/proc/sys/dev/`	设备参数（特定设备）
`/proc/sys/firmware/`	firmware 加载
`/proc/sys/abi/`	应用二进制接口（x86 等架构）
`/proc/sys/sunrpc/`	Sun RPC（NFS 客户端）

A.6 `/proc/self/` 特殊接口

`/proc/self/` 是 `/proc/[current_pid]/` 的快捷方式：

路径	用途
`/proc/self/exe`	当前进程的可执行文件
`/proc/self/cwd`	当前工作目录
`/proc/self/environ`	当前进程的环境变量
`/proc/self/status`	当前进程的状态
`/proc/self/maps`	当前进程的内存映射
`/proc/self/fd/`	当前进程的文件描述符
`/proc/self/ns/`	当前进程的 namespace

A.7 性能关键 `/proc` 接口

路径	主要内容	性能影响
`/proc/stat`	系统级 CPU 统计	低
`/proc/loadavg`	平均负载	极低
`/proc/meminfo`	内存统计	极低
`/proc/pressure/cpu`	CPU 压力（PSI）	极低
`/proc/pressure/memory`	内存压力（PSI）	极低
`/proc/pressure/io`	I/O 压力（PSI）	极低
`/proc/[pid]/io`	进程 I/O	中（频繁读取需优化）
`/proc/[pid]/stat`	进程状态	中
`/proc/net/tcp`	TCP 连接	中-高（连接数大时）
`/proc/[pid]/maps`	进程内存映射	高（大型进程）

A.8 安全相关 sysctl 速查

```bash
# 内核级加固
kernel.yama.ptrace_scope = 1           # 限制 ptrace
kernel.randomize_va_space = 2          # 完整 ASLR
kernel.kptr_restrict = 2               # 隐藏内核符号
kernel.dmesg_restrict = 1              # 限制 dmesg
kernel.unprivileged_bpf_disabled = 2   # 禁用非特权 BPF
```

```
kernel.modules_disabled = 0             # 模块加载开关（按需启用）
kernel.usermodehelper.binfmt_misc = 1   # CVE-2025-31133 缓解
kernel.kexec_load_disabled = 1          # 禁用 kexec
kernel.sysrq = 0                        # 禁用 SysRq
kernel.perf_event_paranoid = 3          # 限制 perf

# 虚拟内存
vm.mmap_min_addr = 65536                # 防止空指针 mmap

# 文件系统保护
fs.protected_hardlinks = 1
fs.protected_symlinks = 1
fs.protected_fifos = 2
fs.protected_regular = 2
```

0.9 附录 B：可复现实验脚本

本附录提供 12 个实验脚本，覆盖本书所有关键概念。建议在测试环境（VM / 容器）上运行，不要在生产环境。

0.9.1 B.1 实验环境准备

```
# 实验 1: 准备一个实验 VM
# 推荐: Ubuntu 24.04 + Linux 6.6 内核
# 资源: 2 CPU / 4 GB RAM / 20 GB disk

# 实验环境检查脚本
#!/usr/bin/env bash
# /usr/local/bin/lab_env_check.sh

set -euo pipefail
echo "=== 实验环境检查 ==="

KERNEL=$(uname -r)
echo "[*] 内核版本: $KERNEL"
if [[ ! "$KERNEL" =~ ^6\. ]]; then
    echo "[!] 建议使用 Linux 6.x 内核"
fi

echo "[*] /proc 挂载选项: "
findmnt /proc

echo "[*] 关键 sysctl: "
for k in kernel.yama.ptrace_scope kernel.randomize_va_space \
         kernel.kptr_restrict kernel.dmesg_restrict \
         kernel.unprivileged_bpf_disabled; do
    v=$(sysctl -n $k 2>/dev/null || echo "n/a")
    echo "  $k = $v"
done

echo "[*] 工具检查: "
for cmd in python3 bpftrace bcc bpftool node_exporter prometheus \
           promtail loki grafana-server falco auditd; do
    if command -v $cmd > /dev/null; then
        echo "  ✓ $cmd"
    else
        echo "  ☒ $cmd (未安装)"
    fi
done
```

```
echo ""
echo "[*] 准备 OK"
```

0.9.2 B.2 实验 1：/proc/[pid] 进程信息探索

```
#!/usr/bin/env bash
# /usr/local/bin/lab_proc_pid.sh
# 目标：理解 /proc/[pid]/* 的结构和权限

set -euo pipefail

echo "=== 实验 1：进程信息探索 ==="
echo ""

# 准备：启动一个后台进程
echo "[Step 1] 启动一个 demo 进程"
DEMO_CMD='while true; do echo $RANDOM; sleep 1; done'
bash -c "$DEMO_CMD" > /tmp/demo_output.log 2>&1 &
DEMO_PID=$!
echo "demo PID: $DEMO_PID"
sleep 1

echo ""
echo "[Step 2] 看 /proc/$DEMO_PID 目录结构"
ls -la /proc/$DEMO_PID/

echo ""
echo "[Step 3] 看 /proc/$DEMO_PID/status"
cat /proc/$DEMO_PID/status

echo ""
echo "[Step 4] 看 /proc/$DEMO_PID/cmdline"
cat /proc/$DEMO_PID/cmdline | tr '\0' ' '
echo ""

echo "[Step 5] 看 /proc/$DEMO_PID/cwd 符号链接"
ls -la /proc/$DEMO_PID/cwd

echo "[Step 6] 看 /proc/$DEMO_PID/exe 符号链接"
ls -la /proc/$DEMO_PID/exe

echo ""
echo "[Step 7] 看 /proc/$DEMO_PID/environ（隐藏敏感信息）"
cat /proc/$DEMO_PID/environ | tr '\0' '\n'

echo ""
echo "[Step 8] 看 /proc/$DEMO_PID/maps"
cat /proc/$DEMO_PID/maps | head -20

echo ""
echo "[Step 9] 看 /proc/$DEMO_PID/fd 列表"
ls -la /proc/$DEMO_PID/fd/

echo ""
echo "[Step 10] 看 fd 4（stdout）指向哪里"
readlink /proc/$DEMO_PID/fd/1

echo ""
echo "[Cleanup] 杀 demo 进程"
kill $DEMO_PID
wait $DEMO_PID 2>/dev/null || true
```

```
echo " 完成"
```

0.9.3 B.3 实验 2: hidepid 的实际效果

```
#!/usr/bin/env bash
# /usr/local/bin/lab_hidepid.sh
# 目标: 验证 hidepid=1 / hidepid=2 对其他用户的实际限制

set -euo pipefail

echo "=== 实验 2: hidepid 限制 ==="
echo ""

# 准备两个用户
if ! id testuser1 > /dev/null 2>&1; then
    useradd -m testuser1
fi
if ! id testuser2 > /dev/null 2>&1; then
    useradd -m testuser2
fi

echo "[Step 1] 当前 /proc 挂载选项"
findmnt /proc

echo ""
echo "[Step 2] 不启用 hidepid 时: testuser2 能读 testuser1 的 cmdline 吗? "
sudo -u testuser2 cat /proc/$(pgrep -u testuser1 -f 'sleep 60')/cmdline 2>&1 | head -c 100
echo ""

echo ""
echo "[Step 3] 启用 hidepid=2, testuser2 还能读吗? "
mount -o remount,hidepid=2,gid=procmon /proc
findmnt /proc

# 准备一个 demo 进程
bash -c 'while true; do sleep 1; done' > /dev/null 2>&1 &
DEMO_PID=$!
chown testuser1:testuser1 /proc/$DEMO_PID  # 不可行: procfs 的 owner 是内核
# 改用: 让 demo 进程以 testuser1 身份跑
sudo -u testuser1 bash -c 'while true; do sleep 1; done' > /dev/null 2>&1 &
DEMO_PID=$!
sleep 1

echo "testuser2 试读 testuser1 的 cmdline: "
sudo -u testuser2 cat /proc/$DEMO_PID/cmdline 2>&1 | head -c 100
echo ""

echo ""
echo "[Step 4] 恢复原状"
mount -o remount,hidepid=0 /proc  # 实际上 remount 不能去掉 hidepid, 需要重启
echo " (hidepid 需要重启才能完全去掉) "

# 清理
kill $DEMO_PID 2>/dev/null || true
```

0.9.4 B.4 实验 3: /proc/net/tcp 的连接反查

```
#!/usr/bin/env python3
# /usr/local/bin/lab_net_tcp_invert.py
# 目标: 从 /proc/net/tcp 还原所有 ESTABLISHED 连接的「PID + 进程名 + 远端」

import os
```

```
import socket
import struct

TCP_STATE = {
    "01": "ESTABLISHED",
    "02": "SYN_SENT",
    "03": "SYN_RECV",
    "04": "FIN_WAIT1",
    "05": "FIN_WAIT2",
    "06": "TIME_WAIT",
    "07": "CLOSE",
    "08": "CLOSE_WAIT",
    "09": "LAST_ACK",
    "0A": "LISTEN",
    "0B": "CLOSING",
}

def parse_hex_addr(hex_str):
    """hex_ip:hex_port → ip:port"""
    ip_hex, port_hex = hex_str.split(":")
    # IPv4: 8 hex chars, reversed byte order
    if len(ip_hex) == 8:
        ip = ".".join(
            str(int(ip_hex[i:i+2], 16))
            for i in (6, 4, 2, 0)
        )
    else:
        # IPv6: 32 hex chars
        ip = socket.inet_ntoa(
            struct.pack("<LL",
                        int(ip_hex[0:8], 16),
                        int(ip_hex[8:16], 16))
            if len(ip_hex) == 16
            else b'\x00' * 4
        )
    port = int(port_hex, 16)
    return ip, port

def parse_proc_net_tcp(path):
    """ 解析 /proc/net/tcp[/tcp6]"""
    connections = []
    with open(path) as f:
        next(f)  # skip header
        for line in f:
            parts = line.split()
            if len(parts) < 10:
                continue
            local_hex, rem_hex, st, _, _, _, _, _, _, inode = parts[:10]
            if st not in TCP_STATE:
                continue
            local_ip, local_port = parse_hex_addr(local_hex)
            rem_ip, rem_port = parse_hex_addr(rem_hex)
            connections.append({
                "state": TCP_STATE[st],
                "local": f"{local_ip}:{local_port}",
                "remote": f"{rem_ip}:{rem_port}",
                "inode": int(inode),
            })
    return connections

def find_pid_by_inode(inode):
    """ 从 /proc/*/fd/* 找 inode 对应的 PID"""
```

```
    for entry in os.listdir("/proc"):
        if not entry.isdigit():
            continue
        fd_dir = f"/proc/{entry}/fd"
        try:
            for fd in os.listdir(fd_dir):
                link = os.readlink(f"{fd_dir}/{fd}")
                if link == f"socket:[{inode}]":
                    return int(entry)
        except (PermissionError, FileNotFoundError, ProcessLookupError):
            continue
    return None

def get_comm(pid):
    try:
        with open(f"/proc/{pid}/comm") as f:
            return f.read().strip()
    except (PermissionError, FileNotFoundError):
        return "?"

def main():
    print("=== 当前所有 ESTABLISHED 连接（按 PID 聚合）===")
    print(f"{'PID':<8} {'COMM':<20} {'LOCAL':<32} {'REMOTE':<32}")
    print("-" * 100)

    seen_pids = set()
    for path in ["/proc/net/tcp", "/proc/net/tcp6"]:
        if not os.path.exists(path):
            continue
        for conn in parse_proc_net_tcp(path):
            if conn["state"] != "ESTABLISHED":
                continue
            pid = find_pid_by_inode(conn["inode"])
            if pid is None:
                comm = "?"
                pid_str = "?"
            else:
                if pid in seen_pids:
                    continue
                seen_pids.add(pid)
                comm = get_comm(pid)
                pid_str = str(pid)
            print(f"{pid_str:<8} {comm:<20} {conn['local']:<32} {conn['remote']:<32}")

if __name__ == "__main__":
    main()
```

跑起来你会看到类似：

```
=== 当前所有 ESTABLISHED 连接（按 PID 聚合）===
PID      COMM                 LOCAL                            REMOTE
1234     sshd                 192.168.1.10:22                  192.168.1.100:54321
5678     nginx                0.0.0.0:80                       192.168.1.100:43120
9012     postgres             192.168.1.20:5432                192.168.1.30:51234
...
```

0.9.5 B.5 实验 4：/proc/sys/kernel/core_pattern 提权演示（仅限实验环境）

```
#!/usr/bin/env bash
# /usr/local/bin/lab_core_pattern_privesc.sh
# 目标：演示 CVE-2025-52881 风格的 core_pattern 提权
# 警告：仅在隔离 VM 里跑，会永久修改 core_pattern
```

```
set -euo pipefail

echo "=== 实验 4: core_pattern 提权演示 ==="
echo "[!] 警告: 本实验会修改 /proc/sys/kernel/core_pattern"
echo "[!] 完成后必须恢复"
echo ""

ORIG_PATTERN=$(cat /proc/sys/kernel/core_pattern)
echo "[*] 原 core_pattern: $ORIG_PATTERN"

# 准备一个 setuid 程序（或者一个能改 core_pattern 的能力）
# 简化: 用 root 权限直接演示
BACKDOOR=/tmp/lab_backdoor
cat > $BACKDOOR.c << 'EOF'
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("[!] Backdoor executed as uid=%d euid=%d\n", getuid(), geteuid());
    // 实际攻击中: 反弹 shell / 植入 cron / 写 /root/.ssh/authorized_keys
    return 0;
}
EOF
gcc -o $BACKDOOR $BACKDOOR.c
chmod 755 $BACKDOOR

echo "[*] 写入恶意 core_pattern"
echo "|$BACKDOOR" > /proc/sys/kernel/core_pattern
cat /proc/sys/kernel/core_pattern

echo ""
echo "[*] 让一个进程 crash 触发 core dump"
# 让 shell crash
sleep 30 &
PID=$!
sleep 0.5
kill -SIGSEGV $PID 2>/dev/null || true
sleep 2

echo ""
echo "[*] 应该看到 backdoor 被执行: "
ls -la core.* 2>/dev/null || echo "（无 core 文件，说明 backdoor 已执行）"

echo ""
echo "[Cleanup] 恢复 core_pattern"
echo "$ORIG_PATTERN" > /proc/sys/kernel/core_pattern
rm -f $BACKDOOR $BACKDOOR.c core.* /tmp/lab_backdoor.c
echo "[*] 完成"
```

0.9.6 B.6 实验 5: eBPF 监控 /proc 访问

```
#!/usr/bin/env bash
# /usr/local/bin/lab_ebpf_proc_trace.sh
# 目标: 用 bpftrace 实时监控所有 /proc 访问

set -euo pipefail

if ! command -v bpftrace > /dev/null; then
    echo "[!] bpftrace 未安装"
    echo "    Ubuntu/Debian: apt-get install -y bpftrace"
    echo "    CentOS/RHEL:    dnf install -y bpftrace"
    exit 1
```

```
fi

echo "=== 实验 5: eBPF 监控 /proc 访问 ==="
echo "[*] 启动 bpftrace"
echo "[*] 在另一个终端跑: ls /proc/$$/status"
echo "[*] Ctrl-C 退出"
echo ""

# 监控 vfs_read 的所有调用，过滤 /proc 路径
bpftrace -e '
kprobe:vfs_read {
    $file = (struct file *)arg0;
    $dentry = $file->f_path.dentry;
    $name = $dentry->d_name.name;
    $parent = $dentry->d_parent->d_name.name;

    // 只关心 /proc 路径
    if ($parent[0] >= "0" && $parent[0] <= "9") {
        // 父目录是 PID，可能是 /proc/[pid]/xxx
        printf("pid=%d comm=%s read /proc/%s/%s\n",
               pid, comm, $parent, $name);
    }
}
'
```

0.9.7 B.7 实验 6: Prometheus 自定义 /proc exporter

```
# 把第 4.4.1 节的 proc_io_exporter.py 部署起来

# /usr/local/bin/install_proc_io_exporter.sh
#!/usr/bin/env bash
set -euo pipefail

echo "=== 部署 proc_io_exporter ==="

# 安装 prometheus_client
pip3 install prometheus_client || pip install prometheus_client

# 复制脚本
cp /usr/local/bin/proc_io_exporter.py /usr/local/bin/ 2>/dev/null || true

# 创建 systemd unit
cat > /etc/systemd/system/proc-io-exporter.service << 'EOF'
[Unit]
Description=/proc I/O Prometheus Exporter
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/proc_io_exporter.py
Restart=on-failure
User=procmon
Group=procmon
NoNewPrivileges=true
ProtectSystem=strict
ProtectHome=true
PrivateTmp=true

[Install]
WantedBy=multi-user.target
EOF
```

```
# 创建用户
useradd -r -s /usr/sbin/nologin procmon 2>/dev/null || true

# reload + start
systemctl daemon-reload
systemctl enable proc-io-exporter
systemctl start proc-io-exporter

sleep 2
systemctl status proc-io-exporter

echo ""
echo "[*] 测试"
curl -s http://localhost:9878/metrics | head -20
```

0.9.8 B.8 实验 7: auditd 规则测试

```
#!/usr/bin/env bash
# /usr/local/bin/lab_auditd_proc.sh
# 目标: 验证 auditd 能否捕获 /proc 访问

set -euo pipefail

if ! command -v auditctl > /dev/null; then
    echo "[!] auditd 未安装"
    echo "    apt-get install -y auditd"
    exit 1
fi

echo "=== 实验 7: auditd 监控 /proc/[pid]/environ ==="

# 加载规则
echo "[Step 1] 加载 audit 规则"
auditctl -w /proc/[0-9]*/environ -p r -k proc-environ-read
auditctl -l | grep proc-environ-read

echo ""
echo "[Step 2] 触发 environ 读取"
sleep 30 &
PID=$!
sleep 0.5
cat /proc/$PID/environ | head -c 50
kill $PID 2>/dev/null || true
sleep 1

echo ""
echo "[Step 3] 查 audit 日志"
ausearch -k proc-environ-read --interpret | head -20

echo ""
echo "[Cleanup] 删除规则"
auditctl -W /proc/[0-9]*/environ -p r -k proc-environ-read
echo "[*] 完成"
```

0.9.9 B.9 实验 8: hidepid 全套配置

```
#!/usr/bin/env bash
# /usr/local/bin/lab_hidepid_full.sh
# 目标: 完整配置 hidepid, 包括采集器例外

set -euo pipefail
```

```
echo "=== 实验 8: hidepid 完整配置 ==="

# 创建采集器组
echo "[Step 1] 创建 procmon 组"
groupadd -f procmon

# 把采集器用户加入组
for user in node_exporter prometheus falco; do
    if id $user > /dev/null 2>&1; then
        usermod -aG procmon $user
        echo "  ✓ $user 加入 procmon"
    fi
done

echo ""
echo "[Step 2] 配置 /etc/fstab"
# 注意: 备份原 /proc 行
if ! grep -q "hidepid" /etc/fstab; then
    # 找原 /proc 行
    PROC_LINE=$(grep "/proc" /etc/fstab | head -1)
    echo " 原配置: $PROC_LINE"

    # 注释原行, 加新行
    sed -i 's|^[^#]*\s*/proc\s*|#&\nproc /proc proc defaults,hidepid=2,gid=procmon 0 0|' /etc/fstab
    echo " 新配置:"
    grep -E "^proc\s+/proc" /etc/fstab
fi

echo ""
echo "[Step 3] 重启或 remount"
echo " 推荐重启。如果不能重启, 可手动 remount: "
echo "  mount -o remount,hidepid=2,gid=procmon /proc"

echo ""
echo "[Step 4] 验证"
echo "  cat /proc/1/cmdline  # 应该 Permission denied"
echo "  sudo -u node_exporter cat /proc/1/cmdline  # 应该可读"
```

0.9.10 B.10 实验 9: /proc/sys/kernel/yama/ptrace_scope 验证

```
#!/usr/bin/env bash
# /usr/local/bin/lab_ptrace_scope.sh
# 目标: 验证 ptrace_scope 对进程调试的影响

set -euo pipefail

echo "=== 实验 9: ptrace_scope 验证 ==="

# 当前值
echo "[Step 1] 当前 yama.ptrace_scope: "
sysctl kernel.yama.ptrace_scope

echo ""
echo "[Step 2] 切换到不同值, 看 gdb 行为"
for scope in 0 1 2 3; do
    sysctl -w kernel.yama.ptrace_scope=$scope > /dev/null
    echo ""
    echo "--- scope = $scope ---"
    if [ "$scope" = "0" ]; then
        echo "  行为: classic 模式, 任何同 uid 进程都可 ptrace"
    elif [ "$scope" = "1" ]; then
        echo "  行为: restricted 模式, 仅父子进程"
```

```
    elif [ "$scope" = "2" ]; then
        echo "  行为: admin only, 仅 CAP_SYS_PTRACE"
    elif [ "$scope" = "3" ]; then
        echo "  行为: 完全禁用"
    fi
done

# 恢复
sysctl -w kernel.yama.ptrace_scope=1 > /dev/null
echo ""
echo "[*] 恢复为 scope=1"
```

0.9.11 B.11 实验 10: 综合应急响应演练

```
#!/usr/bin/env bash
# /usr/local/bin/lab_ir_dr.sh
# 目标: 完整跑一次 /proc 相关的应急响应

set -euo pipefail

echo "=== 实验 10: 综合应急响应演练 ==="
echo ""
echo "[!] 这是一个完整的演练剧本"
echo ""

# 场景: 某台机器被检测到异常 /proc/[pid]/environ 读取
# 你的任务: 按剧本完成响应

read -p " 按 Enter 开始演练..."

echo ""
echo "[T+0] 收到告警"
echo "  alert: ProcEnvironRead"
echo "  host: web-prod-01"
echo "  pid: 12345"
echo "  exe: /usr/local/bin/suspicious"
echo "  uid: 1000"

read -p " 按 Enter 继续..."

echo ""
echo "[T+5] Step 1: 隔离进程 (不杀, 留着取证) "
echo "  sudo kill -STOP 12345"
echo "  → 进程被暂停, 但保留内存供取证"

read -p " 按 Enter 继续..."

echo ""
echo "[T+10] Step 2: 取证 dump"
echo "  sudo gcore 12345            # 内存 dump"
echo "  sudo cat /proc/12345/cmdline | tr '\0' ' '"
echo "  sudo cat /proc/12345/status"
echo "  sudo cat /proc/12345/maps | head -50"
echo "  sudo ls -la /proc/12345/fd/"

read -p " 按 Enter 继续..."

echo ""
echo "[T+15] Step 3: 杀进程 + 网络隔离"
echo "  sudo kill -9 12345"
echo "  sudo iptables -A OUTPUT -m owner --uid-owner 1000 -j DROP"
```

```
read -p " 按 Enter 继续..."

echo ""
echo "[T+20] Step 4: 影响范围评估"
echo "  sudo ausearch -k proc-environ-read --interpret | tail -100"
echo "  sudo ausearch --start yesterday --end now --interpret | \\"
echo "    grep -E 'core_pattern|modules'"

read -p " 按 Enter 继续..."

echo ""
echo "[T+30] Step 5: 通知 + 工单"
echo "  → wecom 通知 oncall 群"
echo "  → 创建 P1 Jira"
echo "  → 邮件发给安全主管"

echo ""
echo "[*] 演练完成"
echo ""
echo "[反思] 这个剧本里，哪些步骤可以自动化？"
echo "       哪些步骤是「人不可替代」的？"
echo "       oncall 手册里应该写哪几条？"
```

0.9.12 B.12 实验 11: 自定义 auditd 规则并验证

```
#!/usr/bin/env bash
# /usr/local/bin/lab_auditd_deploy.sh
# 目标: 部署一套完整的 /proc auditd 规则

set -euo pipefail

echo "=== 实验 11: 完整 auditd 规则部署 ==="

RULES_FILE="/etc/audit/rules.d/10-proc-deep.rules"

cat > $RULES_FILE << 'EOF'
# /proc 纵深防御 audit 规则集
# Last updated: 2026

# --- 凭证相关 ---
-w /proc/[0-9]*/environ -p r -k proc-environ-read
-w /proc/[0-9]*/cmdline -p r -k proc-cmdline-read

# --- 内存相关 ---
-a always,exit -F arch=b64 -S process_vm_readv -k proc-mem-read
-a always,exit -F arch=b64 -S ptrace -F a0=0x4 -k proc-ptrace-read

# --- 内存布局发现 ---
-w /proc/[0-9]*/maps -p r -k proc-maps-read
-w /proc/[0-9]*/smaps -p r -k proc-smaps-read
-w /proc/[0-9]*/smaps_rollup -p r -k proc-smaps-read

# --- fd 列表 ---
-a always,exit -F arch=b64 -S getdents64 -F dir=/proc/[0-9]*/fd -k proc-fd-list

# --- 内核符号 ---
-w /proc/kallsyms -p r -k proc-kallsyms-read
-w /proc/kcore -p r -k proc-kcore-read
-w /proc/modules -p r -k proc-modules-read

# --- 网络相关 ---
-w /proc/net/tcp -p r -k proc-net-tcp-read
```

```
-w /proc/net/tcp6 -p r -k proc-net-tcp-read
-w /proc/net/udp -p r -k proc-net-udp-read

# --- 内核参数写入（高危）---
-w /proc/sys/kernel/core_pattern -p wa -k proc-core-pattern-write
-w /proc/sys/kernel/modules_disabled -p wa -k proc-modules-write
-w /proc/sys/kernel/yama/ptrace_scope -p wa -k proc-ptrace-scope-write
-w /proc/sys/kernel/kptr_restrict -p wa -k proc-kptr-restrict-write
-w /proc/sys/kernel/dmesg_restrict -p wa -k proc-dmesg-restrict-write
-w /proc/sys/kernel/unprivileged_bpf_disabled -p wa -k proc-bpf-disable-write
-w /proc/sys/kernel/sysrq -p wa -k proc-sysrq-write

# --- 排除已知 agent ---
EOF

# 排除已知 agent（用 -F pid!=）
for proc in node_exporter prometheus falco; do
    pids=$(pgrep $proc 2>/dev/null || true)
    for pid in $pids; do
        echo "-a never,exit -F pid=$pid" >> $RULES_FILE
    done
done

echo "[*] 规则文件: $RULES_FILE"
cat $RULES_FILE

echo ""
echo "[*] 加载规则"
augenrules --load

echo ""
echo "[*] 验证"
auditctl -l | grep proc- | head -10

echo ""
echo "[*] 测试触发"
bash -c 'while true; do sleep 1; done' > /dev/null 2>&1 &
TEST_PID=$!
sleep 0.5
cat /proc/$TEST_PID/environ > /dev/null
sleep 1

echo ""
echo "[*] 应该看到 audit 记录"
ausearch -k proc-environ-read --interpret | tail -5

kill $TEST_PID 2>/dev/null || true
echo ""
echo "[*] 完成"
```

0.9.13 B.13 实验 12：自定义 Grafana dashboard 导入

```
#!/usr/bin/env bash
# /usr/local/bin/lab_grafana_import.sh
# 目标：把第 7.2.3 节的 dashboard 导入 Grafana

set -euo pipefail

GRAFANA_URL="${GRAFANA_URL:-http://localhost:3000}"
GRAFANA_USER="${GRAFANA_USER:-admin}"
GRAFANA_PASS="${GRAFANA_PASS:-admin}"
```

```
echo "=== 实验 12: 导入 Grafana dashboard ==="

# 创建 dashboard JSON
cat > /tmp/proc-overview.json << 'EOF'
{
  "dashboard": {
    "title": "/proc 可观测性概览",
    "panels": [
      {
        "title": " 进程状态",
        "type": "stat",
        "gridPos": {"h": 4, "w": 6, "x": 0, "y": 0},
        "targets": [
          {
            "expr": "node_processes_state",
            "legendFormat": "{{state}}"
          }
        ]
      },
      {
        "title": "TCP 连接状态",
        "type": "timeseries",
        "gridPos": {"h": 8, "w": 12, "x": 0, "y": 4},
        "targets": [
          {"expr": "proc_tcp_state_established", "legendFormat": "ESTABLISHED"},
          {"expr": "proc_tcp_state_listen", "legendFormat": "LISTEN"},
          {"expr": "proc_tcp_state_time_wait", "legendFormat": "TIME_WAIT"}
        ]
      },
      {
        "title": "I/O 热点 Top 10",
        "type": "table",
        "gridPos": {"h": 8, "w": 6, "x": 12, "y": 4},
        "targets": [
          {
            "expr": "topk(10, sum by (comm) (rate(proc_io_write_bytes_total[5m])))",
            "format": "table",
            "instant": true
          }
        ]
      }
    ]
  }
}
EOF

echo "[*] 导入到 Grafana..."
curl -X POST -H "Content-Type: application/json" \
  -u "$GRAFANA_USER:$GRAFANA_PASS" \
  -d @/tmp/proc-overview.json \
  "$GRAFANA_URL/api/dashboards/db"

echo ""
echo "[*] 完成。访问 Grafana 查看"
```

0.9.14 B.14 实验工具：统一实验管理脚本

```
#!/usr/bin/env bash
# /usr/local/bin/lab_all.sh
# 跑全套实验（实验 1-12）

set -euo pipefail
```

```
echo "================================================"
echo "        /proc 攻防演义 - 实验全套"
echo "================================================"
echo ""

SCRIPTS_DIR=/usr/local/bin

for script in \
    lab_env_check \
    lab_proc_pid \
    lab_hidepid \
    lab_net_tcp_invert \
    lab_ebpf_proc_trace \
    lab_auditd_proc \
    lab_hidepid_full \
    lab_ptrace_scope \
    lab_ir_dr \
    lab_auditd_deploy \
    lab_grafana_import; do

    if [ -x "$SCRIPTS_DIR/$script.sh" ]; then
        echo ""
        echo "================================================"
        echo " 实验: $script"
        echo "================================================"
        "$SCRIPTS_DIR/$script.sh"
        read -p " 按 Enter 继续下个实验 (Ctrl-C 退出) ..."
    fi
done

echo ""
echo "================================================"
echo "                全套完成"
echo "================================================"
```

0.9.15 B.15 故障排查速查表

症状	可能原因	排查步骤
node_exporter 抓不到进程指标	`hidepid=2` 让非 root 看不到	把 node_exporter 加入 `procmon` 组
gdb attach 失败	`yama.ptrace_scope` 限制	临时设 0，调试完改回 1
auditd 不记录	规则没加载	`auditctl -l` 验证
bpftrace 报错	内核不支持或权限不够	`uname -r`，`sudo`
Prometheus 抓取失败	exporter 进程死了	`systemctl status proc-io-exporter`
Loki 收不到 audit 日志	Promtail 配置错	`journalctl -u promtail`
Falco 告警风暴	规则过严	audit 模式 + 白名单
容器里看不到 `/proc/1`	runc maskedPath	检查 OCI spec
cgroup v1 vs v2 不一致	系统混合	升级或统一
eBPF 程序加载失败	内核 verifier	简化代码，检查 map 大小

0.10 附录 C：参考资源

0.10.1 C.1 官方文档

- `man 5 proc` —/proc 接口规范（本地索引，按数字找页）

- `man 2 process_vm_readv` —跨进程读内存
- `man 2 ptrace` —进程跟踪
- `man 2 openat` —`openat(2)` 是 `/proc/[pid]/*` 读取的实际 syscalls
- `man 5 procfs` —procfs 挂载选项 (`hidepid` 等)
- `man 8 sysctl` —`/proc/sys/*` 字段工具
- `man 8 auditd` —审计守护进程
- `man 1 ps` —`/proc` 之上构建的进程列表工具
- Linux Kernel 源码 `fs/proc/` —实现参考（Bootlin 在线浏览）
- Linux 内核官方文档—`/proc` 文件系统
- Linux man-pages 项目主页 —最新版 man-pages 源码

0.10.2 C.2 推荐阅读

- 《Understanding the Linux Kernel, 3rd Ed》—Bovet & Cesati，第 13 章 `/proc`（Amazon）
- 《Linux Device Drivers, 3rd Ed》—Corbet 等，Chapter 6「Advanced Char Driver Operations」（LWN 在线版 ·GitHub 仓库）
- 《The Linux Programming Interface》—Kerrisk，第 12 章「System and Process Information」（出版社主页 ·No Starch 出版）
- 《Linux Observability with BPF》—Gregg，第 4-6 章（GitHub 配套代码）
- 《eBPF 编程指南》—Cilium 官方电子书（含中英文版）

0.10.3 C.3 工具下载

指标 / 可观测性 - node_exporter —Prometheus 官方 Linux 指标导出器 - Prometheus —指标采集与告警 - Grafana —可视化与 dashboard

eBPF 工具链 - BCC（BPF Compiler Collection）—Python/C 高级 eBPF 编程框架 - bpftrace —eBPF 高级 DSL，shell 一行写 trace - bpftool —内核自带（Debian/Ubuntu: `linux-tools-common`），BPF 程序/map 管理 - libbpf —C 库，BCC 替代品，CO-RE 友好 - Pixie —New Relic 开源的 eBPF 可观测性平台

运行时安全 - Falco —CNCF Incubating，基于 eBPF/kmod 的运行时检测 - Tetragon —Cilium 团队出品，能 hook 后直接 Sigkill - Tracee —Aqua Security 出品，eBPF runtime security

容器运行时 - runc —OCI 标准运行时（**务必 1.2.8** 修复 CVE-2025-52881）- containerd —CNCF 毕业，工业级容器运行时 - CRI-O —Kubernetes 原生 CRI

审计与日志 - auditd —Linux 原生审计子系统 - Loki —Grafana 团队的轻量日志聚合 - Promtail —Loki 的日志采集 agent - Elasticsearch / ELK —重量级日志聚合

0.10.4 C.4 CVE 索引

按年份倒序排列，每条链到 NVD（NIST 漏洞数据库，含 CVSS 评分、官方描述、引用）：

- CVE-2025-52881 —runc maskedPath 绕过（容器逃逸，需升级 runc 1.2.8）
- CVE-2025-31133 —usermodehelper 调用（内核提权）
- CVE-2025-40271 —`proc_readdir_de()` use-after-free
- CVE-2025-52565 —runc 同系列漏洞
- CVE-2026-46333 —`get_dumpable()` race condition（读 SSH host 私钥）
- CVE-2026-31431 —与 CVE-2022-0847 对照案例
- CVE-2022-2588 —dirty_cred（cred 结构利用）
- CVE-2022-0847 —Dirty Pipe（管道缓冲区覆盖）
- CVE-2022-0185 —legacy_buffers heap overflow
- CVE-2021-4034 —PwnKit（pkexec 本地提权）

补充入口：- CVE.org 主页 —CVE 编号官方源 - NVD 主页 —美国国家漏洞数据库（CVSS + CPE 完整）- Linux Kernel CVE 列表（Ubuntu 跟踪） —Ubuntu 标记的 kernel CVE - GitHub Advisory Database —runc / containerd 等容器相关 advisory

0.10.5 C.5 进一步阅读（公众号、博客）

- 《SSH 暴力破解深度防御》—同系列姊妹篇

- 《Linux 进程调试实战》—ptrace / gdb / strace 与 `/proc` 的关系
- 《eBPF 入门到精通》—eBPF 生态全景

英文技术博客 - Brendan Gregg’s Blog —BPF / 性能分析领域权威 - LWN.net —Linux 内核深度报道 - Cilium Documentation —eBPF / Tetragon 一手信息 - Aqua Security Tracee 项目 —Tracee 仓库（Aqua 已停更博客，集中发版到仓库）- Sysdig Blog —Falco 起源团队 - Cloudflare Blog —Linux 分类 —生产环境 `/proc` 实战案例

中文社区 - Linux 中国 —中文 Linux 技术资讯 - TLDP 中文版 —Linux 文件系统层级 - 开源工场 / OSC —中文开源技术社区

链接核对说明：以上所有链接在 2026 年 7 月 23 日最后一次校对时均可访问。NVD / CVE.org / man7.org / GitHub 都是长期稳定的源；个别 O’Reilly 页面需登录才能看完整内容，链接到 Amazon / 出版社替代页以便找到书。如果发现死链，请在 GitHub Issue 提单：https://github.com/LeisureLinux/ebook-procfs-hardening/issues