

Phiên 4 — exec và /proc: đọc ngược một tiến trình

Phiên 3 dựng vòng đời tiến trình từ phía người viết chương trình. Phiên này đi ngược: cho một PID lạ đang chạy, moi ra bằng hết những gì hệ thống biết về nó — và biết file nào trong /proc đang nói dối.

THỜI LƯỢNG	THUỘC	MỐC	ĐẦU VÀO
2 giờ, không lẩn giờ	Phase 1 — Process, fd, signal	chuẩn bị cho M-L1 (Phiên 8)	Phiên 3 đã xong
ARTIFACT			
hồ sơ một tiến trình thật, một trang			

Kỹ năng của phiên này dùng được ngay ở công việc thật: một tiến trình đang ăn CPU, đang giữ file, đang chạy sai phiên bản — và bạn chỉ có PID của nó. /proc trả lời được gần hết, với điều kiện biết file nào đáng tin và file nào chỉ là thứ tiến trình *muốn* bạn thấy.

QUAN HỆ VỚI PHIÊN 3

Phiên 3 đã xử lý fork, wait và encoding của exit status, nên phiên này không lặp lại. Phần exec ở đây đi xuống tầng kernel: ai quyết định một file được chạy thế nào. Phần còn lại là /proc.

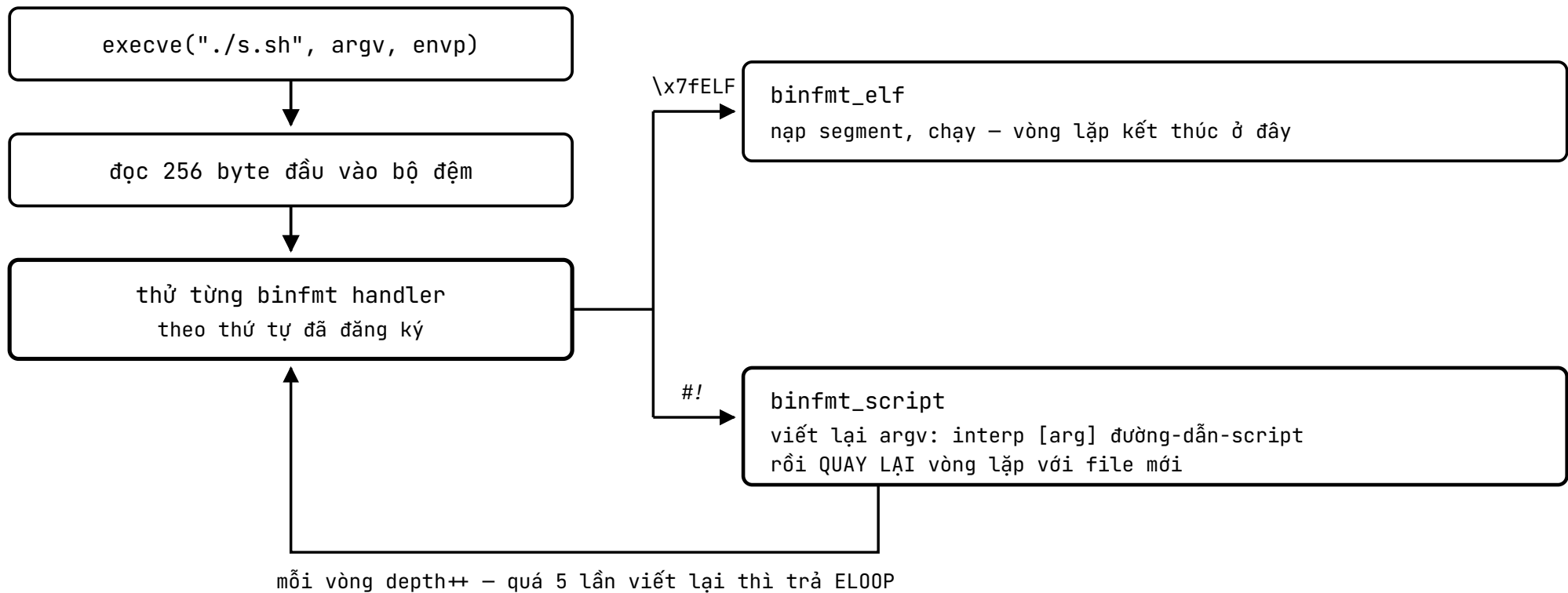
MỤC 00

Trước khi bấm giờ

- ☐ Chạy trong guest, như mọi phiên của Phase 1. Snapshot trước khi bắt đầu.
- ☐ gcc, strace, pstree vẫn chạy được (đã kiểm ở Phiên 3).
- ☐ Trong guest có ít nhất một service thật đang chạy để mở ở Block B. Không cần cài gì lạ — systemd-resolved, cron, sshd đều đủ phức tạp. Chọn một cái không phải phiên ssh của chính bạn, để không tự cắt chân mình.
- ☐ Ghi vào note: cat /proc/sys/kernel/yama/ptrace_scope, ulimit -n, ulimit -Hn, sysctl fs.nr_open fs.file-max. Bốn con số này là đề bài của Block C.
- ☐ Ghi uname -r và getconf PAGESIZE. Cái sau cần cho statm.
- ☐ Kiểm stat -fc %T /sys/fs/cgroup — phải ra cgroup2fs. Không phải thì mọi thứ về /proc/<pid>/cgroup ở dưới sẽ khác, và Phase 2 với Phase 4 cũng khác.

Kernel quyết định chạy file bằng cách nào

`execve` nhận một đường dẫn, không nhận một "loại file". Việc phân loại xảy ra sau: kernel mở file, đọc **256 byte đầu** vào một bộ đệm, rồi đưa lần lượt cho từng *binary format handler* đã đăng ký cho tới khi có cái nhận.



Script không phải "loại file thứ hai" – nó là một lần viết lại tham số rồi chạy lại đúng vòng lặp đó. Đó là lý do interpreter của script có thể lại là script.

HANDLER	NHẬN FILE NÀO	KẾT THÚC VÒNG LẶP?
binfmt_elf	bắt đầu bằng \x7fELF	có — nạp segment và chạy
binfmt_script	bắt đầu bằng #!	không — viết lại argv rồi quay lại
binfmt_misc	khớp magic hoặc phần mở rộng đã đăng ký qua /proc/sys/fs/binfmt_misc/	không — chèn interpreter rồi quay lại

BẦY · COMMENT TRONG KERNEL KHÔNG KHỚP VỚI CHÍNH CODE CỦA NÓ

Giới hạn đệ quy nằm ở vòng lặp trong `fs/exec.c`. Comment ngay trên nó viết *"This allows 4 levels of binfmt rewrites before failing hard"*, còn điều kiện là `if (depth > 5) return -ELOOP;`. Đo thực tế cho **4 lần** "interpreter tự nó lại là script" — khớp man page, lệch comment. Lỗi trả về là `ELOOP` từ Linux 3.8; trước đó là `ENOEXEC`.

Bài học dùng lại được: comment trong source là ý định của người viết ở một thời điểm, không phải hành vi. Khi hai thứ lệch nhau thì code đúng, và phép đo mới chốt.

Một dòng shebang, mở ra từng đoạn

Toàn bộ luật gói trong một hàm ngắn của kernel, và nó khắt khe hơn hầu hết người dùng tưởng.

MỘT DÒNG SHEBANG

```
#!/[ ]/usr/bin/env -S python3 -u[ ]
```



- 1 · hai byte đầu tiên của FILE. Một khoảng trắng đứng trước là ENOEXEC.
- 2 · space/tab ở đây bị bỏ qua, bao nhiêu cũng được.
- 3 · đường dẫn interpreter. Bị cắt cụt ⇒ ENOEXEC, không chạy nhằm file.
- 4 · dấu phân cách; nhiều khoảng trắng liên tiếp cũng chỉ tính là một.
- 5 · TẤT CẢ phần còn lại là MỘT tham số. Khoảng trắng bên trong giữ nguyên.
- 6 · space/tab cuối dòng bị cắt bỏ.

Đoạn 5 là chỗ hay hỏng nhất khi mang script từ hệ khác sang: Linux không tách tham số theo khoảng trắng, nên `#!/usr/bin/env python3 -u` đi tìm một chương trình tên đúng là `python3 -u`.

Trần độ dài. Kernel chỉ đọc 256 byte đầu, nên dòng shebang dùng được tối đa **255 byte tính cả `#!`**. Trước Linux 5.1 con số là 127.

Cắt cụt không còn im lặng. Cho tới Linux 4.20, đường dẫn interpreter quá dài bị cắt rồi vẫn đem đi mở — nghĩa là có thể chạy nhầm một file khác. Từ **Linux 5.0**, kernel kiểm rằng đường dẫn không bị cắt, không đảm bảo được thì trả `ENOEXEC`. Phần *tham số* thì vẫn bị cắt im lặng.

`argv[0]` bạn truyền vào bị vớt. Với script, kernel dựng lại `argv` thành `interpreter [arg]` `đường-dẫn-script arg...` — man page nói thẳng: *"there is no way to get the `argv[0]` that was passed to the `execve()` call"*.

ENOENT khi file rõ ràng tồn tại

CHỖ TÀI LIỆU PHỔ BIẾN CHẨN ĐOÁN SAI

Chạy `./run.sh` và shell in ra `./run.sh: No such file or directory` trong khi file nằm sờ sờ ở đó. Thông báo đúng, chỉ là nó không nói về file bạn nghĩ.

`execve` trả `ENOENT` khi *"The file pathname or a script or ELF interpreter does not exist"*. Kernel đọc dòng `#!`, lấy nguyên chuỗi làm đường dẫn, mở không được thì trả lỗi đó ra ngoài. Nguyên nhân số một: file có ngắt dòng **CRLF**, nên interpreter mà kernel đi tìm tên là `/bin/bash` kèm một ký tự `\r` ở cuối.

Phân biệt với `EACCES`, vốn là chuyện quyền: script không có bit execute, interpreter không có bit execute, hoặc filesystem mount `noexec`. Cách chẩn đoán nhanh nhất là nhìn thẳng vào byte đầu file thay vì đoán — xem spoiler Block A.

BLOCK A • ~30 PHÚT

Một lần exec, bóc từng lớp

Việc cần làm — phần quan sát:

- Viết một script một dòng chỉ có shebang trỏ tới `/bin/echo` và không làm gì khác. Chạy nó với vài tham số. Giải thích **chính xác** thứ tự các chuỗi mà `echo` in ra — thứ tự đó là bằng chứng cho luật viết lại `argv`.
- Sửa shebang thành dạng có tham số (`#!/bin/echo -x -y`) rồi chạy lại. Đếm xem `echo` nhận được **mấy** tham số từ dòng shebang. Con số này là điểm chính của cả block.
- Tạo chuỗi đệ quy: script A có shebang trỏ tới script B, B trỏ tới C, và cứ thế. Tăng dần cho tới khi `execve` gãy. **Ghi lại `errno` và con số chính xác**, rồi so với con số trong man page.
- Dựng lại bẫy CRLF một cách có chủ đích: lấy một script chạy được, đổi ngắt dòng sang CRLF, chạy lại. Ghi thông báo lỗi. Rồi tìm cách *nhìn thấy* nguyên nhân bằng một lệnh, không phải bằng suy đoán.
- Chạy một script Python (hoặc bất kỳ interpreter nào có sẵn) và đọc ba giá trị `comm`, `cmdline`, `exe` của tiến trình đó khi nó còn sống. Ba giá trị này khác nhau — đây là đầu vào cho mục 02.

Chương trình 4 – phần phải tự viết: một chương trình C gọi `execv` để chạy một lệnh có sẵn (ví dụ `/bin/cat`) nhưng truyền `argv[0]` là một chuỗi bịa hoàn toàn.

- Trả lời trước khi gõ: kernel có kiểm tra `argv[0]` khớp với tên file không? Man page dùng từ gì để mô tả ràng buộc đó – và từ đó có nghĩa là bắt buộc hay không?
- Chạy xong, kiểm tra ba chỗ: `ps` hiển thị gì, `/proc/<pid>/comm` ra gì, `/proc/<pid>/exe` trỏ đâu. Một trong ba nói thật.
- Câu hỏi để tự trả lời và ghi vào note: nếu một tiến trình lạ trên máy production đang giả dạng tên khác trong `ps`, bạn dùng cái gì để biết nó thật sự là chương trình nào?

■ HIỆN LỆNH QUAN SÁT · BLOCK A

NHÌN THẮNG VÀO BYTE ĐẦU FILE

TRONG GUEST

```
# ba cach, deu tra loi cung mot cau hoi
$ head -c 64 ./run.sh | od -c | head -3
$ file ./run.sh                # bao "with CRLF line terminators" neu dinh
$ sed -n '1p' ./run.sh | cat -A # CRLF hien ra thanh ^M$

# chua CRLF
$ sed -i 's/\r$//' ./run.sh
```

XEM KERNEL DỰNG LẠI ARGV THẾ NÀO

TRONG GUEST

```
$ printf '#!/bin/echo\n' > s1.sh && chmod +x s1.sh
$ ./s1.sh A B C

$ printf '#!/bin/echo -x -y\n' > s2.sh && chmod +x s2.sh
$ ./s2.sh A B

# tang syscall: xem execve nhan gi va tra ve gi
$ strace -f -e trace=execve,execveat ./s2.sh A B

# errno tho, khong qua loi dien giai cua shell
$ strace -f -e trace=execve ./run.sh 2>&1 | grep -E 'execve|ENOENT|EACCES|ELOOP'
```

BA CÁI TÊN CỦA MỘT TIẾN TRÌNH

TRONG GUEST

```
# chay mot script interpreter roi doc no trong luc con song
$ printf '#!/usr/bin/python3\nimport time; time.sleep(300)\n' > s.py
$ chmod +x s.py && ./s.py foo bar &
$ P=$!

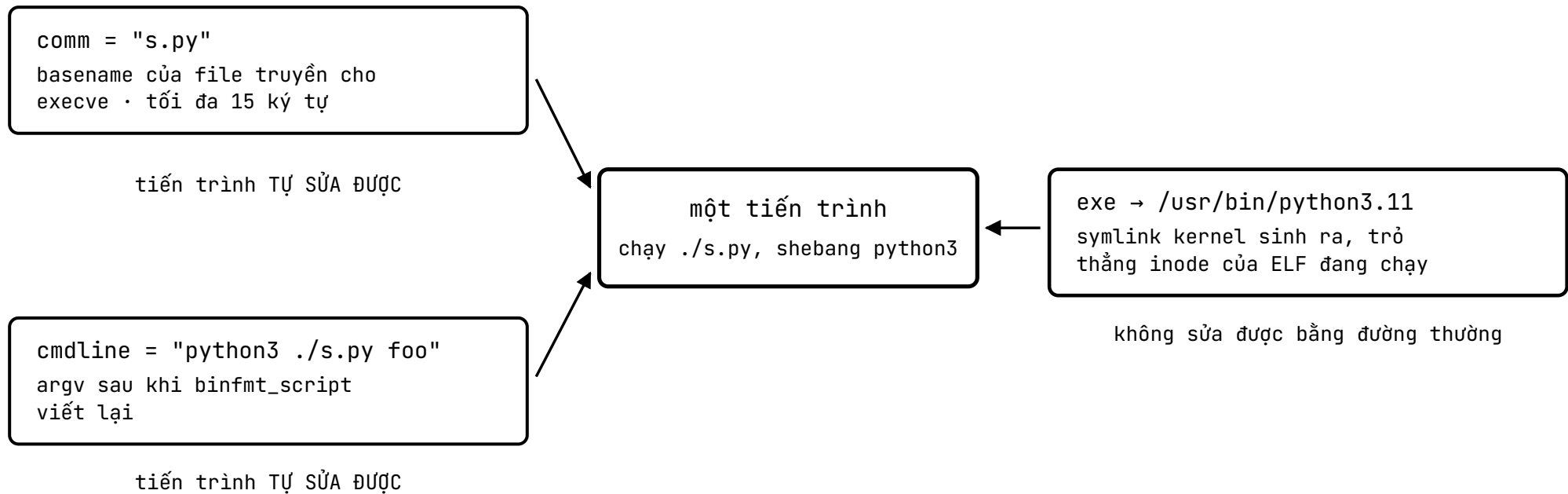
$ cat /proc/$P/comm
$ tr '\0' ' ' < /proc/$P/cmdline ; echo
$ readlink /proc/$P/exe
$ kill $P
```

Nghiệm thu Block A

- ☐ Giải thích được thứ tự argv mà interpreter nhận, bằng luật chứ không bằng "chạy thử thấy vậy".
- ☐ Con số thật của giới hạn đệ quy interpreter, kèm errno, đo trên chính guest của mình.
- ☐ Một lệnh chẩn đoán CRLF ghi vào note – thứ sẽ dùng lại nhiều lần trong đời.
- ☐ Ba giá trị `comm` / `cmdline` / `exe` của cùng một tiến trình script, và một câu về việc cái nào đáng tin.

MỤC 02

Ba cái tên của một tiến trình



Ba giá trị, ba câu chuyện, cùng một tiến trình. Hai cái bên trái là thứ tiến trình *muốn* bạn thấy; chỉ cái bên phải do kernel giữ.

NGUỒN	ĐẶT TỪ ĐÂU	TIẾN TRÌNH TỰ SỬA?	GIỚI HẠN
/proc/<pid >/comm	basename của <i>tên file</i> truyền cho <code>execve</code>	có — <code>prctl(PR_SET_NAME)</code> , ghi thẳng vào file, hoặc API đặt tên thread	16 byte kể cả NUL ⇒ 15 ký tự , cắt cụt im lặng
/proc/<pid >/cmdline	<code>argv</code> — với script là bản kernel đã viết lại	có — ghi đè vùng nhớ <code>argv</code> tại chỗ	phân tách bằng NUL; rỗng với zombie và kernel thread
/proc/<pid >/exe	symlink tới inode của ELF <i>cuối cùng</i> được nạp	không	đọc cần quyền <code>ptrace-read</code>

Man page của `cmdline` tự tóm tắt gọn hơn mọi cách diễn đạt khác: *“Think of this file as the command line that the process wants you to see.”* Cùng logic áp cho `comm`. Đây là lý do `ps` không phải công cụ điều tra, và cũng là lý do một binary đa chức năng như `busybox` đổi hành vi theo `argv[0]` mà không cần chép file.

BẦY · VÌ SAO PKILL KHÔNG BẮT ĐƯỢC TIẾN TRÌNH

`comm` chỉ giữ 15 ký tự. Công cụ khớp theo tên tiến trình mặc định khớp trên `comm` đã cắt cụt, nên một daemon tên dài sẽ trượt. Khớp trên `cmdline` thì trúng — nhưng khi đó bạn đang khớp trên chuỗi mà *tiến trình tự chọn*, và có thể trúng nhầm cả tiến trình khác đang có chuỗi đó trong dòng lệnh, kể cả chính lệnh bạn vừa gõ. Hai cách khớp, hai kiểu sai khác nhau.

BLOCK B · ~35 PHÚT

Hồ sơ một tiến trình thật

ĐÂY LÀ ARTIFACT CỦA PHIÊN

Chọn **một** service đang chạy trong guest và viết một trang hồ sơ về nó, mỗi mục kèm đường dẫn `/proc` đã dùng để lấy. Mục tiêu không phải chép output ra file — mà là với mỗi dòng, nói được nó *nghĩa là gì*.

Hồ sơ phải trả lời được các câu sau:

CÂU HỎI	LẤY Ở ĐÂU	NÓI ĐƯỢC THÊM ĐIỀU GÌ
Nó thật sự là chương trình nào?	<code>exe</code> , <code>comm</code> , <code>cmdline</code>	ba giá trị có khớp nhau không, lệch thì vì sao
Ai sinh ra nó, nó đang ở trạng thái gì?	<code>status</code>	<code>PPid</code> , <code>State</code> , <code>Threads</code> , <code>TracerPid</code>
Nó khởi động với môi trường nào?	<code>environ</code>	và vì sao đây <i>không</i> phải môi trường hiện tại — xem mục 03
Nó đang đứng ở thư mục nào, gốc filesystem của nó là gì?	<code>cwd</code> , <code>root</code>	<code>root</code> khác / nghĩa là gì
Nó dùng bao nhiêu bộ nhớ — con số nào cộng lại được?	<code>status</code> , <code>smaps_rollop</code>	RSS so với PSS, và vì sao chỉ một trong hai cộng được
Nó bị chặn bởi giới hạn nào?	<code>limits</code>	soft và hard của <code>NOFILE</code> , so với <code>ulimit</code> trong shell của bạn
Nó nằm ở đâu trong cây cgroup?	<code>cgroup</code>	vì sao dòng của cgroup v2 luôn bắt đầu bằng <code>0::</code>

CÂU HỎI	LẤY Ở ĐÂU	NÓI ĐƯỢC THÊM ĐIỀU GÌ
Nó đang giữ những gì?	fd/	đếm số fd, phân loại file thường / socket / pipe
Nó ở chung namespace với bạn không?	ns/	so với /proc/self/ns/ — cái nào giống, cái nào khác
Bộ nhớ của nó trông thế nào?	maps	nhận ra [heap], [stack], [vdso], và các vùng ánh xạ từ file

Hai câu bắt buộc phải có trong hồ sơ, vì chúng là chỗ phân biệt người đọc `/proc` với người chép `/proc`:

- Một chỗ mà **hai file khác nhau nói hai chuyện lệch nhau** về cùng một tiến trình, kèm giải thích vì sao cả hai đều đúng.
- Một con số mà bạn **không** đưa vào hồ sơ vì đã kiểm ra rằng nó không đáng tin — kèm lý do.

■ HIỆN LỆNH QUAN SÁT • BLOCK B

Lệnh rời để tra cú pháp. Việc chọn cái nào, đọc ra sao và viết thành hồ sơ là bài tập.

CHỌN MỤC TIÊU VÀ ĐỊNH DANH

TRONG GUEST

```
$ systemctl show --value -p MainPID cron.service
$ P=<pid>

$ readlink /proc/$P/exe ; cat /proc/$P/comm
$ tr '\0' '\n' < /proc/$P/cmdline
$ readlink /proc/$P/cwd ; readlink /proc/$P/root
$ grep -E '^(Name|State|Tgid|Pid|PPid|TracerPid|Uid|Gid|Threads):' /proc/$P/status
```

BỘ NHỚ, GIỚI HẠN, CGROUP

TRONG GUEST

```
$ grep -E '^(VmSize|VmRSS|RssAnon|RssFile|RssShmem|VmSwap):' /proc/$P/status
$ cat /proc/$P/smaps_rollback # Pss là con số cong lai duoc

$ cat /proc/$P/limits
$ prlimit --pid $P # cung du lieu, dinh dang khac

$ cat /proc/$P/cgroup
$ stat -fc %T /sys/fs/cgroup # cgroup2fs = unified v2
```

FD, NAMESPACE, VÙNG NHỚ

TRONG GUEST

```
$ sudo ls -l /proc/$P/fd/ | head -20
$ sudo ls /proc/$P/fd/ | wc -l
$ sudo ls -l /proc/$P/fd/ | grep -c socket

# cung namespace hay khong: so ca (st_dev, st_ino)
$ for n in mnt net pid uts ipc user cgroup; do
    printf '%-7s self=%s target=%s\n' "$n" \
        "$(sudo stat -L -c '%d:%i' /proc/self/ns/$n)" \
        "$(sudo stat -L -c '%d:%i' /proc/$P/ns/$n)"
done
```

```
$ sudo grep -E '\\[(heap|stack|vdso|vvar)\\]' /proc/$P/maps
$ sudo wc -l /proc/$P/maps
```

Nghiệm thu Block B

- ☐ Hồ sơ một trang, mỗi mục có đường dẫn `/proc` đi kèm.
- ☐ Chỉ ra được một chỗ hai file nói lệch nhau, kèm giải thích.
- ☐ Chỉ ra được một con số đã loại vì không đáng tin, kèm lý do.
- ☐ Trả lời được: tiến trình này có ở chung namespace với shell của bạn không, và bạn biết bằng cách nào.

MỤC 03

Bốn file trong `/proc` để đọc nhầm

`environ` — không phải môi trường hiện tại, và cũng không phải bản chụp

Man page nói: file chứa *"the initial environment that was set when the currently executing program was started via `execve(2)`",* và nếu tiến trình gọi `setenv` sau đó thì *"this file will not reflect those changes"*. Đúng — nhưng lý do **không** phải vì kernel đã đông cứng dữ liệu lại.

Kernel không lưu bản sao nào cả. Mỗi lần bạn `read()`, nó **đọc sống** khoảng địa chỉ `[env_start, env_end)` trong bộ nhớ tiến trình. Hai hệ quả nghe có vẻ ngược nhau nhưng cùng đúng:

- `setenv` không hiện ra vì thư viện C cấp phát chuỗi mới trên heap và sửa *mảng con trỏ*, chứ không ghi đè vùng cũ trên stack. Kernel đọc vùng cũ nên thấy giá trị cũ.
- Nội dung **vẫn có thể đổi hoặc trông như rác**, nếu tiến trình ghi đè trực tiếp lên chính vùng đó. Đó là điều mọi daemon đổi tên hiển thị trong `ps` đều làm — nó giẫm lên vùng argv và tràn sang vùng env. Đọc ra cụt hoặc lộn xộn là bình thường, không phải lỗi quyền và không phải tiến trình hỏng.

`statm` — hai trường chết hẳn, hai trường được chính man page dán nhãn không chính xác

Bảy trường, đơn vị là **trang** chứ không phải byte hay kB. Trường 5 và trường 7 *"unused since Linux 2.6; always 0"*; trường 2 và 3 bị đánh dấu *inaccurate*, và man page nêu lý do: *"Some of these values are inaccurate because of a kernel-internal scalability optimization."* Thấy trường dirty bằng 0 rồi kết luận tiến trình không có trang bản là sai — trường đó chết từ 2006, số thật nằm ở `smaps`.

exe và hậu tố (deleted) — dấu hiệu tốt, nhưng không đủ

Khi pathname của binary bị unlink, symlink được nối thêm chuỗi "(deleted)". Đây chính là toàn bộ cơ chế của công cụ báo "service nào cần restart sau khi nâng cấp gói": nó tìm chuỗi đó trong exe và trong maps. Ba điều cần biết trước khi tin nó:

CÁCH FILE BỊ THAY	EXE CÓ HẬU TỐ (DELETED)?
Bị rm	có
Bị rename() đề lên — cách trình quản lý gói cài đặt	có, vì inode cũ mất tên
Bị ghi đè tại chỗ (cp lên file cũ, hoặc >)	không — inode giữ nguyên, tiến trình vẫn chạy code cũ mà không có dấu hiệu nào

Thêm nữa, hậu tố này là chuỗi ký tự nối vào, nên một file có tên thật chứa đúng chuỗi đó sẽ đánh lừa mọi script grep. `proc_pid_maps(5)` nói thẳng điều đó — *"Note that this is ambiguous too."* — còn `proc_pid_exe(5)` thì không nhắc, dù cùng một cơ chế. Cách kiểm chắc chắn hơn là so inode giữa `/proc/<pid>/exe` đã deref và file hiện tại trên đĩa.

limits — thứ duy nhất nói sự thật về giới hạn

Resource limit được kế thừa qua `fork` và giữ nguyên qua `exec`. Nghĩa là giá trị của một tiến trình được quyết định lúc nó được sinh ra, và mọi thay đổi trong file cấu hình sau đó không chạm tới nó. Đây là gốc rễ của tình huống "đã tăng giới hạn rồi mà vẫn lỗi": `ulimit -n` trong shell cho ra số mới, nhưng `/proc/<pid>/limits` của service vẫn ra số cũ.

CHỖ TÀI LIỆU PHỔ BIẾN CHỈ SAI ĐỊA CHỈ

`/etc/security/limits.conf` không áp dụng cho service của systemd. Nó là file cấu hình của một module PAM, và module đó chỉ chạy khi có một PAM session được mở — *"All limit settings are set per login"*, và *"Only the session module type is provided"*. Service do PID 1 khởi chạy không mở PAM session trừ khi unit khai báo tường minh. Chỗ đúng để chỉnh là `LimitNOFILE=` trong unit, hoặc `DefaultLimitNOFILE=` cho toàn hệ.

Và `fs.file-max` không chứa được `Too many open files`. Hai thứ khác trực: `fs.file-max` là trần toàn hệ thống, vượt thì lỗi là `ENFILE`; còn `Too many open files` thường là `EMFILE`, tức đựng `RLIMIT_NOFILE` của riêng tiến trình. Phân biệt bằng `strace`: thấy `EMFILE` thì sửa unit, thấy `ENFILE` mới là chuyện toàn hệ.

Cuối cùng, đừng chép `LimitNOFILE=infinity` từ mạng về. Mặc định hiện tại đã là soft 1024, hard 524288, và tài liệu systemd viết đúng hai chữ *"Don't use."* cho việc tự nâng soft limit trong unit. Soft được giữ ở 1024 có lý do kỹ thuật: `select(2)` không xử lý được fd từ 1024 trở lên, nên nâng soft toàn cục sẽ khiến chương trình vô tình nhận fd số cao rồi hỏng bất thường. Ứng dụng nào cần nhiều fd thì tự nâng soft lên hard khi khởi động — chỉ nó mới biết nó có dùng `select` hay không.

Ai đọc được cái gì

Không phải file nào trong `/proc/<pid>/` cũng cùng một luật quyền. Có ba nhóm, và biết mình đang ở nhóm nào sẽ tiết kiệm rất nhiều lần gõ `sudo` mò mẫm.

NHÓM	FILE	BỊ PTRACE_SCOPE CHẶN?
Không kiểm ptrace	cmdline, comm, status, stat, limits, cgroup	không
ptrace mức <i>read</i>	environ, exe, cwd, root, fd/, maps, ns/, auxv	không — xem cảnh báo dưới
ptrace mức <i>attach</i>	syscall, stack; và mọi thứ strace -p / gdb -p làm	có
Chỉ chủ tiến trình	fdinfo/, mountstats	không liên quan

BẦY · PTRACE_SCOPE KHÔNG CHẶN THỨ MÀ NHIỀU NGƯỜI TƯỞNG

Ubuntu đặt `/proc/sys/kernel/yama/ptrace_scope` bằng 1, và rất nhiều bài viết nói giá trị đó "khóa `/proc` của tiến trình khác". Không đúng. Yama chỉ can thiệp vào thao tác *attach*. Với cùng một user, đọc `environ`, `maps`, `exe`, `fd/` của tiến trình khác vẫn chạy bình thường; thứ bị chặn là `gdb -p`, `strace -p`, và hai file `syscall` với `stack`.

Cách phân biệt tại chỗ: nếu `cat` một file trong `/proc/<pid>/` báo lỗi quyền thì nguyên nhân là user khác hoặc `hidepid`, *không* phải Yama. Còn `strace -p` báo `Operation not permitted` trong khi `cat` vẫn chạy thì đúng là Yama.

BLOCK C · ~20 PHÚT

Ba phép kiểm chứng ngược

Ba thí nghiệm ngắn, mỗi cái bác bỏ một câu người ta hay nói. **Đoán kết quả trước, rồi mới chạy**, và ghi cả hai vào note.

1. "**environ** cho biết biến môi trường hiện tại của tiến trình." Viết một chương trình nhỏ: in PID của chính nó, gọi `setenv` đặt một biến có tên dễ nhận ra, rồi ngủ. Trong lúc nó ngủ, tìm biến đó trong `/proc/<pid>/environ`. Tìm không thấy thì giải thích bằng cơ chế ở mục 03.
2. "**Sửa giới hạn trong file cấu hình là service nhận ngay.**" Chọn một service đang chạy. Ghi lại `NOFILE` soft/hard trong `/proc/<pid>/limits`. Đổi giới hạn ở nơi đúng cho service đó, *chưa* restart, rồi đọc lại. Rồi restart và đọc lần ba. Ba con số này là toàn bộ bài học về "kế thừa lúc sinh".

Câu hỏi tự trả lời: có cách nào đổi giới hạn của một tiến trình *đang chạy* mà không restart không? Nếu có thì cần quyền gì?
3. "**ps** cho biết tiến trình đang chạy chương trình gì." Dùng chương trình 4 của Block A. Trong lúc nó chạy, chứng minh bằng `/proc` rằng `ps` đang hiển thị một cái tên bịa. Rồi làm thí nghiệm ngược: xóa hoặc thay file binary trong lúc một tiến trình đang chạy nó, và tìm dấu vết trong `/proc`.

Nghiem thu Block C

- ☐ Ba cặp "đoán / thực tế" trong note. Đoán đúng cả ba nghĩa là bài quá dễ — thêm một biến thể lạ hơn.
- ☐ Ba con số `NOFILE` theo đúng thứ tự trước-sửa / sau-sửa-chưa-restart / sau-restart.

- ☐ Một câu trả lời cho "tiến trình lạ đang giả tên, làm sao biết nó là gì" – bằng lệnh cụ thể.

MỤC 04

Bắt đầu

Đọc nhiều file trong `/proc/<pid>/` không cho một ảnh nhất quán

Cơ chế: không có khoá nào giữ tiến trình đứng yên giữa hai lệnh `cat`. Kernel doc nói thẳng về trường hợp nặng nhất: *"reading /proc/PID/maps or /proc/PID/smaps is inherently racy (consistent output can be achieved only in the single read call)"*. Ngay cả một file cũng chỉ nhất quán trong phạm vi một lời gọi `read()`.

Hệ quả: mọi phép tính chéo giữa `status`, `maps` và `smaps` đều có thể mâu thuẫn, và mâu thuẫn đó không phải bug. Nếu tiến trình chết giữa chừng, lệnh đọc trả lỗi thay vì trả dữ liệu cụt – đó là hành vi đúng.

`smaps` của tiến trình lớn tự nó là một phép đo tổn kém

Với JVM hay trình duyệt, file này dài hàng chục nghìn dòng và việc đọc nó gây tải thật. Có `smaps_rollup` từ Linux 4.14, cho sẵn tổng của mọi vùng. Monitoring lặp lại phải dùng bản rollup.

Và con số đáng cộng là PSS, không phải RSS: *"the count of pages it has in memory, where each page is divided by the number of processes sharing it"*. Cộng RSS của nhiều tiến trình là đếm trùng thư viện dùng chung – đây chính là câu hỏi sẽ quay lại ở Phiên 9.

`/proc/<pid>/cgroup` hiện đường dẫn tương đối với namespace của người đọc

Trong container, file này có thể hiện `/` trong khi tiến trình thật ra nằm sâu trong cây cgroup của host. Và trên hệ hybrid, thấy một dòng `0::` rồi kết luận "thuần v2" là sai – phải kiểm bằng `stat -fc %T /sys/fs/cgroup`.

Bảng tra nhanh

TRIỆU CHỨNG	KIỂM CÁI NÀY TRƯỚC
<code>./script</code> : No such file or directory mà file có thật	Dòng shebang. Nghi CRLF trước tiên: file hoặc <code>od -c</code> vào 64 byte đầu.
Permission denied khi chạy script	Bit execute của <i>script</i> , rồi của <i>interpreter</i> , rồi cờ noexec của mount point.
Shebang có nhiều tham số mà interpreter nhận sai	Linux gộp tất cả thành một tham số. Dùng <code>env -S</code> nếu cần tách.

TRIỆU CHỨNG	KIỂM CÁI NÀY TRƯỚC
Exec format error (ENOEXEC)	Đường dẫn interpreter dài quá 255 byte cả dòng, hoặc dòng <code>#!</code> rỗng, hoặc file không phải định dạng nào kernel nhận.
Too many levels of symbolic links khi chạy script	Không phải symlink – là chuỗi interpreter lồng nhau quá sâu.
<code>pkill <tên></code> không bắt được gì	Tên dài quá 15 ký tự nên <code>comm</code> đã bị cắt. Khớp trên <code>cmdline</code> , và cần thận trọng nhầm.
<code>ps</code> hiện tên lạ hoặc tên vô nghĩa	<code>readlink /proc/<pid>/exe</code> . Chỉ nó do kernel giữ.
Too many open files	<code>strace</code> xem là <code>EMFILE</code> hay <code>ENFILE</code> . <code>EMFILE</code> ⇒ <code>/proc/<pid>/limits</code> và <code>unit</code> , không phải <code>limits.conf</code> .
Đã sửa giới hạn mà tiến trình vẫn dùng số cũ	Limit kế thừa lúc sinh. Chưa restart thì chưa đổi – hoặc dùng <code>prlimit</code> sửa nóng.
<code>strace -p</code> báo không đủ quyền mà <code>cat /proc/<pid>/maps</code> vẫn chạy	Đúng là <code>ptrace_scope</code> . Hai file <code>syscall</code> và <code>stack</code> cũng sẽ bị chặn cùng lúc.
Service chạy code cũ sau khi nâng cấp gói	<code>exe</code> và <code>maps</code> có hậu tố (<code>deleted</code>) không. Không có mà vẫn nghi thì so <code>inode</code> .
Con số bộ nhớ cộng lại lớn hơn RAM thật	Đang cộng RSS. Đổi sang PSS ở <code>smaps_rollup</code> .

MỤC 05

Nghiem thu & câu hỏi cuối phiên

Definition of Done

- ☐ Giải thích được thứ tự `argv` mà interpreter của script nhận, bằng luật.
- ☐ Con số giới hạn đệ quy interpreter đo trên guest của mình, kèm `errno`.
- ☐ Một lệnh chặn đoán CRLF trong note.
- ☐ **Hồ sơ một trang** của một tiến trình thật, mỗi mục kèm đường dẫn `/proc`.
- ☐ Trong hồ sơ có một chỗ hai file nói lệch nhau, và một con số đã bị loại vì không đáng tin.
- ☐ Ba cặp "đoán / thực tế" của Block C.
- ☐ Ba con số `NOFILE` trước-sửa / sau-sửa / sau-restart.
- ☐ Trả lời được bằng lệnh cụ thể: một tiến trình đang giả tên trong `ps`, làm sao biết nó thật sự là chương trình gì.

Hết giờ thì cắt theo thứ tự: Block C trước (lùi sang đầu Phiên 5), rồi các mục cuối của hồ sơ Block B. Block A không cắt được – Phiên 5 về file descriptor dựa thẳng vào việc đọc `/proc/<pid>/fd/` đã quen tay.

Câu hỏi kiểm tra — đóng terminal, đóng trang này

- ① Bạn chạy `./deploy.sh` và shell báo `No such file or directory`, nhưng `ls` thấy file rõ ràng. Nêu hai nguyên nhân có thể và cách phân biệt chúng bằng một lệnh.

■ Ý CẦN CÓ

Nguyên nhân là `ENOENT` nói về *interpreter* trong dòng `#!`, không phải về script. Hai khả năng: đường dẫn interpreter viết sai hoặc chương trình đó chưa cài; hoặc file có ngắt dòng CRLF nên interpreter mà kernel tìm có thêm một ký tự `\r` ở cuối tên. Cách phân biệt: nhìn thẳng byte đầu file (`od -c`, `cat -A`, hoặc `file`) – CRLF hiện ra ngay. Điểm ăn thêm nếu phân biệt được với `EACCES`, vốn là chuyện quyền execute hoặc mount `noexec`.

- ② Script có dòng `#!/usr/bin/env python3 -u`. Trên Linux nó hỏng. Vì sao, và cách sửa?

■ Ý CẦN CÓ

Linux truyền **toàn bộ** phần sau đường dẫn interpreter như một tham số duy nhất, không tách theo khoảng trắng. Nên `env` nhận đúng một chuỗi `"python3 -u"` và đi tìm chương trình mang cái tên đó. Hệ khác có hành vi khác, nên script chép từ nơi khác về hay dính. Cách sửa: dùng `env -S` (cờ sinh ra đúng để tách chuỗi này), hoặc bỏ tham số khỏi shebang và đặt bằng cách khác. Điểm ăn thêm nếu nói được rằng man page khuyến nghị *optional-arg* nên là một từ đơn cho khả chuyển.

- ③ Cho một PID lạ. Kể ba nguồn tên của tiến trình đó, và nói cái nào bạn tin trong một cuộc điều tra sự cố.

■ Ý CẦN CÓ

`comm` (basename của file lúc exec, tối đa 15 ký tự, tiến trình tự đổi được), `cmdline` (argv, tiến trình ghi đè vùng nhớ đó được), `exe` (symlink kernel sinh ra, trỏ thẳng inode ELF đang chạy, không sửa được bằng đường thường). Tin `exe`. Điểm ăn thêm: nói được rằng với script thì ba giá trị lệch nhau một cách hợp lệ – `comm` là tên script, `exe` là interpreter – nên "lệch" không tự động nghĩa là đáng ngờ.

- ④ Một service báo `Too many open files`. Đồng nghiệp đã sửa `/etc/security/limits.conf` và reboot mà vẫn lỗi. Chuyện gì đang xảy ra, và bạn kiểm gì trước?

■ Ý CẦN CÓ

`limits.conf` là cấu hình của một module PAM chạy theo phiên đăng nhập; service do PID 1 khởi chạy không mở phiên PAM nên không bao giờ đọc tới file đó. Chỗ đúng là `LimitNOFILE=` trong unit (hoặc mặc định toàn hệ trong cấu hình của service manager). Kiểm trước: đọc `/proc/<pid>/limits` của chính tiến trình đó – đó là sự thật, khác với mọi file cấu hình. Điểm ăn thêm: phân biệt `EMFILE` (per-process, đúng hướng) với `ENFILE` (toàn hệ, hiếm khi là nguyên nhân trên hệ hiện đại), và biết rằng có cách sửa nóng không cần restart.

- ⑤ `cat /proc/<pid>/environ` của một tiến trình khác vẫn chạy được, nhưng `strace -p` lên chính PID đó thì bị từ chối. Vì sao không mâu thuẫn?

■ Ý CẦN CÓ

Hai thao tác nằm ở hai mức quyền khác nhau. Đọc `environ` thuộc nhóm ptrace mức *read*, còn gắn debugger thuộc mức *attach*. Cơ chế hạn chế của distro (`ptrace_scope`) chỉ can thiệp vào mức *attach*. Điểm ăn thêm nếu kể thêm được hai file cũng bị chặn cùng nhóm với `strace -p` (`syscall`, `stack`), và nếu chỉ ra rằng lỗi quyền khi `cat` thì phải đi tìm nguyên nhân khác – user khác, hoặc tùy chọn mount ẩn `/proc`.

- ⑥ Bạn cộng `VmRSS` của mọi tiến trình lại và ra con số lớn hơn RAM vật lý. Máy có bị gì không?

■ Ý CẦN CÓ

Không. RSS đếm mọi trang tiến trình đang có trong RAM, kể cả trang dùng chung – thư viện chuẩn được đếm lại ở từng tiến trình dùng nó. Con số cộng được là PSS, chia mỗi trang cho số tiến trình đang chia sẻ nó, lấy từ `smaps` hoặc rẻ hơn là `smaps_rollup`. Điểm ăn thêm nếu nhắc rằng đọc `smaps` của tiến trình lớn tự nó đã tốn kém.

Stretch — chỉ khi dư giờ, không tính vào DoD

- **Script hoá hồ sơ Block B.** Biến các bước tay thành một script nhận PID và in ra hồ sơ. Đây là artifact của *bạn*, nên không có lời giải ở đây. Ba tính chất phải đạt: chạy được khi thiếu quyền cho một số file (không được chết giữa chừng), phân biệt được tiến trình đã chết với tiến trình không có quyền đọc, và ghi rõ mọi con số kèm đơn vị.
- `binfmt_misc`. Xem hệ đang đăng ký những gì trong `/proc/sys/fs/binfmt_misc/`, đọc ý nghĩa các cờ, và tự trả lời: cờ nào làm cho một binary khác kiến trúc chạy được bên trong một môi trường cô lập không có sẵn interpreter? Đây là cầu nối sang Phase 4.

- `fdinfo`. Với một socket hoặc một `epoll` fd của service bạn vừa mở, đọc `/proc/<pid>/fdinfo/<fd>` và giải thích từng dòng. Phiên 5 sẽ dựa vào đây.
- **Chạy file qua fd.** `execveat` với `AT_EMPTY_PATH` hoặc `fexecve`: chạy một chương trình mà bạn đang giữ fd, thay vì qua đường dẫn. Tự trả lời: điều đó chặn được race nào mà đường dẫn không chặn được?
- **Rò fd qua exec.** Mở một file không đặt cờ `close-on-exec` rồi `exec` một chương trình khác, và tìm fd đó trong `/proc/<pid>/fd/` của chương trình mới. Rồi thử lại với `close_range`.
- **Ẩn `/proc`.** Đọc tùy chọn mount `hidepid=` và thử đổi trên guest. Chú ý: man page liệt kê thiếu một giá trị đã có trong kernel từ 5.8 – tự đối chiếu với kernel doc.

MỤC 06

Nguồn & đọc thêm

Đọc trong hoặc ngay sau phiên

MAN PAGE

execve(2) <https://man7.org/linux/man-pages/man2/execve.2.html>

Mục "All process attributes are preserved during an execve(), except the following" và đoạn NOTES nói vì sao "exec tạo tiến trình mới" là mô tả sai.

SOURCE / PACKAGING

Linux – fs/binfmt_script.c https://github.com/torvalds/linux/blob/v6.12/fs/binfmt_script.c

Toàn bộ luật shebang nằm trong một hàm `load_script()`. Ba helper `spacetab/next_non_spacetab/next_terminator` trả lời mọi câu hỏi về khoảng trắng.

MAN PAGE

proc_pid_cmdline(5) https://man7.org/linux/man-pages/man5/proc_pid_cmdline.5.html

Câu chốt: "Think of this file as the command line that the process wants you to see."

MAN PAGE

proc_pid_comm(5) https://man7.org/linux/man-pages/man5/proc_pid_comm.5.html

Giới hạn `TASK_COMM_LEN` (16) kể cả `NUL`, cắt cụt im lặng, và là giá trị dùng cho `%e` của `core_pattern`.

MAN PAGE

proc_pid_exe(5) https://man7.org/linux/man-pages/man5/proc_pid_exe.5.html

Symlink tới binary thật; hậu tố "(deleted)" khi pathname đã bị `unlink`. Trang này KHÔNG cảnh báo hậu tố đó nhập nhằng – `maps(5)` mới có.

MAN PAGE

proc_pid_environ(5) https://man7.org/linux/man-pages/man5/proc_pid_environ.5.html

Đọc kỹ câu "will not reflect those changes" và câu về `PR_SET_MM_ENV_START` – hai câu này giải thích hai hiện tượng khác nhau.

MAN PAGE

proc_pid_maps(5) https://man7.org/linux/man-pages/man5/proc_pid_maps.5.html

Sáu cột, các nhãn [heap]/[stack]/[vdso], và câu "Note that this is ambiguous too." về hậu tố (deleted).

MAN PAGE

proc_pid_limits(5) https://man7.org/linux/man-pages/man5/proc_pid_limits.5.html

Từ Linux 2.6.36 mọi user đọc được. Đây là giá trị THẬT đang áp cho tiến trình, khác với file cấu hình.

MAN PAGE

ptrace(2) <https://man7.org/linux/man-pages/man2/ptrace.2.html>

Mục "Ptrace access mode checking": sáu bước, và chỗ LSM (Yama) can thiệp. Đây là luật quyết định file nào trong /proc bạn đọc được.

KERNEL DOC

Kernel doc – Yama LSM <https://docs.kernel.org/admin-guide/LSM/Yama.html>

Bốn mức ptrace_scope. Đọc để thấy nó chỉ chạm tới thao tác ATTACH, không chạm tới READ.

KERNEL DOC

Kernel doc – filesystems/proc.rst <https://docs.kernel.org/filesystems/proc.html>

Nguồn duy nhất định nghĩa PSS đầy đủ, mô tả smaps_rollup, và nói thẳng việc đọc maps/smaps là racy.

MAN PAGE

getrlimit(2), setrlimit(2), prlimit(2) <https://man7.org/linux/man-pages/man2/getrlimit.2.html>

Soft vs hard, câu "for the real user ID of the calling process" ở RLIMIT_NPROC, và prlimit sửa được tiến trình đang chạy.

MAN PAGE

limits.conf(5) / pam_limits(8) <https://man7.org/linux/man-pages/man5/limits.conf.5.html>

Câu "All limit settings are set per login" và "Only the session module type is provided" – đủ để hiểu vì sao file này không luôn được service.

MAN PAGE

systemd.exec(5) <https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.html>

Bảng Limit*=; đọc thẳng ô LimitNOFILE= – upstream viết "Don't use." cho việc tự nâng soft limit.

SÁCH

The Linux Programming Interface – ch. 24–26 <https://man7.org/tlpi/>

Ch. 24 process creation, 25 process termination, 26 monitoring child processes. Đọc theo chương, không đọc tuyến tính.

MAN PAGE

proc_pid_status(5) https://man7.org/linux/man-pages/man5/proc_pid_status.5.html

Định dạng key–value của /proc/PID/status. Đây là đường đọc PPid an toàn, khác hẳn việc cắt trường của stat.

MAN PAGE

strace(1) <https://man7.org/linux/man-pages/man1/strace.1.html>

Đọc -f (theo con), -e trace=, và -T. Ba cờ này đủ cho toàn bộ Phase 1.

Tra khi cần, không đọc tuyến tính

SOURCE / PACKAGING

Linux – fs/exec.c <https://github.com/torvalds/linux/blob/v6.12/fs/exec.c>

Đọc `exec_binprm()`: vòng lặp thử `binfmt` handler, và dòng "if (depth > 5) return -ELOOP". Comment ngay trên nó ghi "4 levels" – lệch với code.

SOURCE / PACKAGING

Linux – include/uapi/linux/binfmts.h <https://github.com/torvalds/linux/blob/v6.12/include/uapi/linux/binfmts.h>

`BINPRM_BUF_SIZE = 256`. Chú ý là file `uapi`, không phải `include/linux/binfmts.h`.

SOURCE / PACKAGING

Linux – fs/proc/base.c <https://github.com/torvalds/linux/blob/v6.12/fs/proc/base.c>

`environ_read()` gọi `access_remote_vm()` – bằng chứng `environ` là đọc sống vùng nhớ, không phải bản chụp lưu sẵn.

MAN PAGE

exec(3) <https://man7.org/linux/man-pages/man3/exec.3.html>

Trang tự tổ chức theo chữ cái l/v/p/e – đọc một lần là hết phải đoán tên hàm nào nhận mảng, hàm nào tra `PATH`.

MAN PAGE

proc_pid_stat(5) https://man7.org/linux/man-pages/man5/proc_pid_stat.5.html

Bảng ký tự trạng thái theo từng kernel version (l có từ 4.14) và thứ tự 52 trường. Xem cả cảnh báo về trường `comm`.

MAN PAGE

proc_pid_smaps(5) https://man7.org/linux/man-pages/man5/proc_pid_smaps.5.html

Danh sách trường mỗi VMA. Trang này thiếu `smaps_rollback` và các trường `Pss_*` mới – bổ sung bằng `kernel doc`.

MAN PAGE

proc_pid_statm(5) https://man7.org/linux/man-pages/man5/proc_pid_statm.5.html

Bảy trường, đơn vị là `TRANG` chứ không phải `byte`, và hai trường chết hửn từ Linux 2.6.

MAN PAGE

proc_sys_fs(5) https://man7.org/linux/man-pages/man5/proc_sys_fs.5.html

Phân biệt `fs.file-max` (toàn hệ, lỗi `ENFILE`) với `fs.nr_open` (trần của `hard RLIMIT_NOFILE`, lỗi `EMFILE`).

MAN PAGE

cgroups(7) <https://man7.org/linux/man-pages/man7/cgroups.7.html>

Mục `/proc/pid/cgroup`: ba trường, và vì sao dòng của v2 luôn bắt đầu bằng "0::".

MAN PAGE

prlimit(1) <https://manpages.ubuntu.com/manpages/noble/man1/prlimit.1.html>

Thuộc `util-linux`, không phải `procps`. Đọc/sửa limit của một PID mà không cần restart.

MAN PAGE

systemd-system.conf(5) <https://www.freedesktop.org/software/systemd/man/latest/systemd-system.conf.html>

DefaultLimitNOFILE= mặc định 1024:524288, và ghi chú rằng limit của PID 1 khác limit truyền cho con.

SOURCE / PACKAGING

needrestart – mã nguồn <https://github.com/liske/needrestart/blob/master/needrestart>

Tìm chuỗi "(deleted)" trong /proc/PID/exe và /proc/PID/maps. Toàn bộ "phép màu" chỉ có vậy.

MAN PAGE

proc(5) <https://man7.org/linux/man-pages/man5/proc.5.html>

Trang gốc, nay chủ yếu là mục lục. Mục MOUNT OPTIONS có hidepid= – nhưng thiếu hidepid=ptraceable.

MAN PAGE

ps(1) <https://man7.org/linux/man-pages/man1/ps.1.html>

Mục PROCESS STATE CODES (D I R S T t W X Z + modifier) và giải thích chuỗi <defunct> nằm ở specifier args.

Để dành cho các phiên sau

KERNEL DOC

Kernel doc – binfmt_misc <https://docs.kernel.org/admin-guide/binfmt-misc.html>

Cú pháp đăng ký và ý nghĩa cờ, đặc biệt cờ F (fix binary) – thứ làm chạy được binary khác kiến trúc trong container.

MAN PAGE

namespaces(7) <https://man7.org/linux/man-pages/man7/namespaces.7.html>

Mục /proc/pid/ns/: so (st_dev, st_ino) của symlink để biết hai tiến trình có cùng namespace không.

MAN PAGE

proc_pid_fdinfo(5) https://man7.org/linux/man-pages/man5/proc_pid_fdinfo.5.html

pos, flags, mnt_id cho mọi fd; thêm dòng riêng cho epoll/inotify/timerfd. Chỉ chủ tiến trình đọc được.

MAN PAGE

proc_pid_mountinfo(5) https://man7.org/linux/man-pages/man5/proc_pid_mountinfo.5.html

File duy nhất trong ba file mount có thông tin propagation (shared:/master:). Sẽ đọc kỹ ở Phase 3.

MAN PAGE

execveat(2) <https://man7.org/linux/man-pages/man2/execveat.2.html>

Linux 3.19. Cùng với fexecve(3) là đường chạy một file qua fd, tránh race đổi file giữa lúc kiểm và lúc chạy.

MAN PAGE

close_range(2) https://man7.org/linux/man-pages/man2/close_range.2.html

Linux 5.9; cờ CLOSE_RANGE_CLOEXEC từ 5.11. Cách sạch để không rò fd qua exec.

pthread_setname_np(3) https://man7.org/linux/man-pages/man3/pthread_setname_np.3.html

Trả ERANGE khi tên quá 16 byte — khác hẳn prctl(PR_SET_NAME) và ghi vào comm, hai cái đó cắt cụt im lặng.

lsuf(8) <https://manpages.ubuntu.com/manpages/noble/en/man8/lsuf.8.html>

Đọc mục +L: "+L1 will select open files that have been unlinked". Dừng lại ở Phiên 13.

Control Group v2 – admin guide <https://docs.kernel.org/admin-guide/cgroup-v2.html>

Mục PID: pids.max, pids.current, pids.events, và câu "fork() will return -EAGAIN". Sẽ đọc kỹ ở Phase 2 và Phase 4.

Ba mẫu đáng nhớ

VÌ SAO 15 KÝ TỰ

Trường giữ tên tiến trình trong kernel dài đúng 16 byte, kể cả byte NUL kết thúc — nên phần hiển thị được là 15. Con số đó không có gì thiêng liêng: nó nằm gọn trong một cache line cùng các trường lân cận, và đã như vậy từ rất lâu. Nhưng nó rò rỉ ra khắp nơi ở tầng trên: `ps` mặc định hiện tên cụt, công cụ khớp theo tên tiến trình trượt mục tiêu, và tên file core dump lấy đúng giá trị cụt đó. Ba API cùng ghi vào trường này lại xử lý tràn theo ba kiểu — hai cái cắt cụt im lặng, một cái trả lỗi. Một hằng số 16 byte, ba hành vi khác nhau, và một lớp bug "tại sao thread của tôi vẫn mang tên cũ".

CON SỐ GIỚI HẠN FILE MÀ KHÔNG AI ĐẶT RA

Mặc định của kernel cho trần `fs.nr_open` là 1048576, một con số tròn trịa. Nhưng trên phần lớn máy Linux hiện đại, đọc ra sẽ là 1073741816 — một con số trông như rác. Nó là dấu vân tay của trình quản lý dịch vụ: lúc khởi động, PID 1 thử ghi giá trị lớn nhất có thể vào `sysctl` đó, kernel từ chối, nó chia đôi rồi thử lại, và giá trị đầu tiên được chấp nhận chính là con số kia. Comment trong source viết thẳng lý do phải làm kiểu mờ: kernel không có API nào cho biết trần thật là bao nhiêu. Bài học nhỏ mà dùng lại được: một con số kỳ quặc trong `sysctl` thường không phải hằng số của ai cả — nó là kết quả của một vòng lặp dò tìm.

CÁI GIÁ CỦA MỘT API CŨ 40 NĂM

Giới hạn mềm số file mở vẫn là 1024 trên gần như mọi hệ Linux hiện đại, dù trần cứng đã là 524288. Lý do không phải hiệu năng mà là `select(2)`: API này dùng một bitmap có kích thước cố định, không biểu diễn được file descriptor từ 1024 trở lên, và hành vi khi vượt là không xác định. Nâng giới hạn mềm cho toàn hệ đồng nghĩa với việc một chương trình bất kỳ có thể vô tình nhận fd số 2000 rồi hỏng theo cách rất khó truy. Nên mặc định giữ nguyên, và tài liệu upstream chuyển trách nhiệm cho ứng dụng: cái nào biết mình không dùng `select` thì tự nâng lên trần cứng khi khởi động. Một quyết định thiết kế của thập niên 1980 vẫn đang định hình giá trị mặc định trên máy bạn hôm nay.

Mọi khẳng định kỹ thuật ở trên đã đối chiếu tài liệu và source upstream — Linux man-pages, kernel source (tag v6.12), kernel Documentation, systemd, linux-pam — tính đến 22/08/2026. Các chỗ lệch với tài liệu phổ biến được đánh dấu riêng kèm bằng chứng. Hai chỗ cần lưu ý về nguồn: comment về giới hạn đệ quy trong kernel không khớp với chính điều kiện ngay dưới nó, và `proc_pid_exe(5)` không cảnh báo về sự nhập nhằng của hậu tố `(deleted)` dù `proc_pid_maps(5)` có.