Filtrowanie wywołań systemowych

Łukasz Sowa

6 kwietnia 2013

O mnie

- Student informatyki @ WEiTI PW
- Programista z zamiłowania i zawodu
- Obszary działalności
  - Hackowanie jądra
  - Bezpieczeństwo
  - Programowanie współbieżne
  - Języki programowania
- Blog techniczny: `http://lukaszsowa.pl/`
- Kontakt: `kontakt@lukaszsowa.pl`

Full disclosure: to mój pierwszy raz...

Agenda

1. Co to są wywołania systemowe?
2. Czym jest filtrowania wywołań systemowych?
3. Korzyści i zastosowania filtrowania
4. Typowe problemy
5. Dostępne implementacje
6. Podsumowanie

Wywołania systemowe 1/2

- Interfejs pomiędzy procesem a jądrem
- Praktycznie jedyny sposób interakcji procesu z systemem
- Ukryte pod funkcjami bibliotecznymi np. `open()`, `getpid()`, `fork()`, `kill()`
- Występują w każdym sensownym programie
- Zbiór dostępnych dla procesu wywołań systemowych precyzyjnie definiuje jego możliwości

Wywołania systemowe 2/2

`strace -cf`

Proces	L. wywołań
`pwd`	36
`ls ~/`	91
`ping -c 20 10.sesja.linuksowa.pl`	545
`ls -la ~/`	720
`find linux-3.8.5`	79679
`vlc ACDC/` $\sim$ 5 min.	261976
`firefox` $\sim$ 5 min.	1927228
`make -C linux-3.8.5`	11007388

Filtrowanie wywołań systemowych

Filtrowanie wywołań systemowych (ang. syscall interposition) to **mechanizm bezpieczeństwa**:

- Pozwala śledzić wykonywane przez proces wywołania i ew. ich argumenty
- Podejmuje na tej podstawie odpowiednie akcje
  - Zezwolenie/zabronienie wykonania wywołania
  - Zabicie procesu
  - Zmiana argumentów wywołania
  - ...

Korzyści

- Ścisła kontrola uprawnień procesu
- Ochrona przed skutkami wstrzykiwania kodu
- Ochrona przed błędnymi danymi wejściowymi
- Ograniczanie zasobów
- Globalne wyłączenie pewnych usług jądra

Zastosowania

- Lekka wirtualizacja (kontenery)
- Serwery usług
- Obliczenia rozproszone
- Aplikacje (nadmiernie) uprzywilejowane
- Element konstrukcyjny sandboksów

Dla kogo?

Kto powinien wykorzystywać filtrowanie wywołań?

- Administratorzy
- Programiści

Zwykli użytkownicy — tylko pośrednio.

Potencjalne problemy

A więc filtrujmy, hurra! Ale...

- Jak to zrobić?
- Czy warto implementować własny mechanizm?
- Skąd wiedzieć które wywołania są potrzebne?
- *devfs*, *procfs*, *debugfs*, *sysfs*...
- Co z *vsyscall* i *vDSO*?
- Przenośność?

Które wywołania blokować? 1/4

- Wiedza na temat programu
- `strace`
- Metoda prób i błędów, ew. uczenie

Które wywołania blokować? 2/4

Wiedza na temat programu

```c
#include <unistd.h>

int main()
{
    const char str[] = "Witaj Sesjo Linuksowa!";
    write(0, str, sizeof(str));
    return 0;
}
```

```
$ gcc -Wall main.c
$ strace ./a.out # ???
```

Które wywołania blokować? 3/4

Wiedza na temat programu

```
$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 35 vars */]) = 0
brk(0)                                  = 0x12d6000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=178042, ...}) = 0
mmap(NULL, 178042, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f19f1dc5000
close(3)                                = 0
open("/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\33\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2035539, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f19f1dc4000
mmap(NULL, 3853456, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f19f1823000
mprotect(0x7f19f19c7000, 2093056, PROT_NONE) = 0
mmap(0x7f19f1bc6000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a3000) = 0x7f19f1bc6000
mmap(0x7f19f1bcc000, 15504, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f19f1bcc000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f19f1dc3000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f19f1dc2000
arch_prctl(ARCH_SET_FS, 0x7f19f1dc3700) = 0
mprotect(0x7f19f1bc6000, 16384, PROT_READ) = 0
mprotect(0x7f19f1df1000, 4096, PROT_READ) = 0
munmap(0x7f19f1dc5000, 178042)          = 0
write(0, "Witaj Sesjo Linuksowa!\0", 23Witaj Sesjo Linuksowa!) = 23
exit_group(0)                           = ?
+++ exited with 0 +++
```

24 wywołania łącznie, 13 różnych!

Nie wszystko widać od razu.

Które wywołania blokować? 4/4

strace

`strace -cf`

Proces	L. wywołań	L. różnych
`pwd`	36	13
`ls ~/`	91	21
`ping -c 20 10.sesja.linuksowa.pl`	545	34
`ls -la ~/`	720	27
`find linux-3.8.5`	79679	18
`vlc ACDC/` $\sim$ 5 min.	261976	79
`firefox` $\sim$ 5 min.	1927228	106
`make -C linux-3.8.5`	11007388	65

Analiza jest... trudna i czasochłonna.

devfs, procfs, debugfs, sysfs...

- Wygodne interfejsy manipulacji systemem
- Obsługiwane zwykle wyłącznie za pomocą `open()`, `read()`, `write()`, `close()`
- Nie ma potrzeby wykonywania innych wywołań
- Utrudnia to właściwe filtrowanie

vsyscall i vDSO

- Służą przyspieszeniu niektórych wywołań...
- ...poprzez omijanie jądra...
- ...i wykonywanie w przestrzeni użytkownika
- Ciężko nadzorować ich wykonanie
- Np. `gettimeofday()`, `getcpu()`

Przenośność

Specyficzne dla każdej architektury:

- Sposoby przechodzenia do trybu jądra
- Liczba wywołań
  x86 350, x86–64 313, ARM 380, Alpha 506
- Numery wywołań
  x86 42: `pipe()`, x86–64 42: `connect()`
- Obsługa *vsyscall*/*vDSO*

Co jest istotne przy wyborze?

- Bezpieczeństwo
- Możliwości i łatwość użycia
- Wydajność

Spoiler: można mieć tylko dwie cechy z trzech.

ptrace 1/2

- Rodzina mechanizmów oparta na `ptrace()`
- Architektura master–slave
- Polityka dostępu w procesie nadzorującym
- Np. SUBTERFUGUE

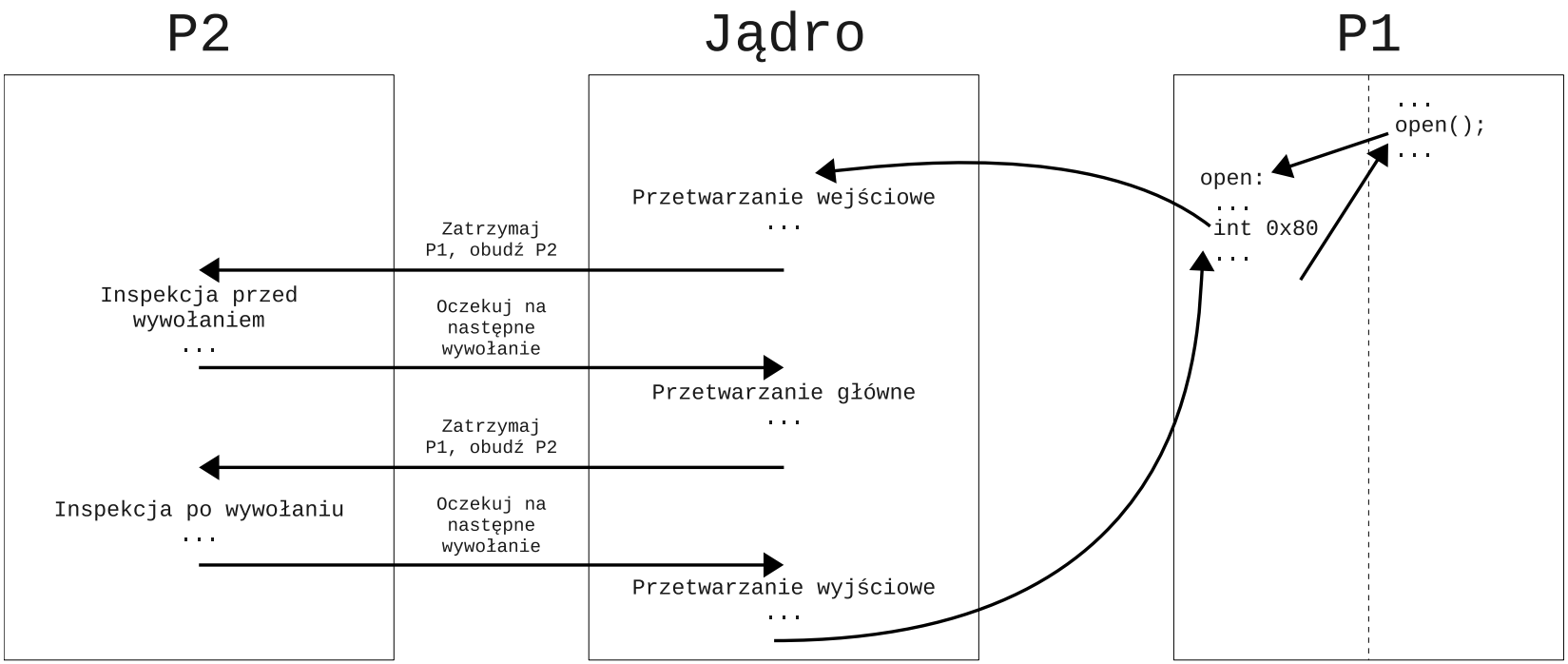
Zalety:

- Dowolna polityka bezpieczeństwa

Wady:

- Narzuty wynikające z przekazywania sterowania
- `ptrace()`

ptrace 2/2


P2
Jądro
P1
...
open();
...
open:
...
int 0x80
...
Przetwarzanie wejściowe
...
Zatrzymaj
P1, obudź P2
Inspekcja przed
wywołaniem
...
Oczekuj na
następne
wywołanie
Przetwarzanie główne
...
Zatrzymaj
P1, obudź P2
Inspekcja po wywołaniu
...
Oczekuj na
następne
wywołanie
Przetwarzanie wyjściowe
...

systrace

- Podobne do `ptrace()` (master–slave)
- Wykorzystuje dodatkowego daemona
- Proste zasady weryfikowane w jądrze
- Bardziej skomplikowane — w daemonie (IPC)

Zalety:

- Automatyczne uczenie polityki dostępu
- Działa na Linuksie, BSD, Mac OS X

Wady:

- Wysokie narzuty wynikające z IPC (do 30%)
- Odkryte liczne błędy bezpieczeństwa
- Nierozwijany od 2009 (?)

Capabilities

- „Parcelują” uprawnienia roota
- Pośrednio mogą służyć jako prosty filtr wywołań

Zalety:

- Zerowy narzut
- Dojrzałość

Wady:

- Tylko dla uprawnień roota
- Czasem niewystarczająca ziarnistość
- Mało rozpowszechniony (wciąż!)

audit subsystem

- Podsystem pozwala na śledzenie wywołań
- Oparty o daemon `auditd`
- Do użycia z mechanizmem *MAC* (?)

Zalety:

- Wydajność
- Dojrzałość

Wady:

- Właściwie tylko logowanie
- Dosyć skomplikowany w konfiguracji

LD_PRELOAD

- `LD_PRELOAD` — zmienna środowiskowa
- Przesłanianie funkcji z *glibc*
- Np. Plash, Ostia

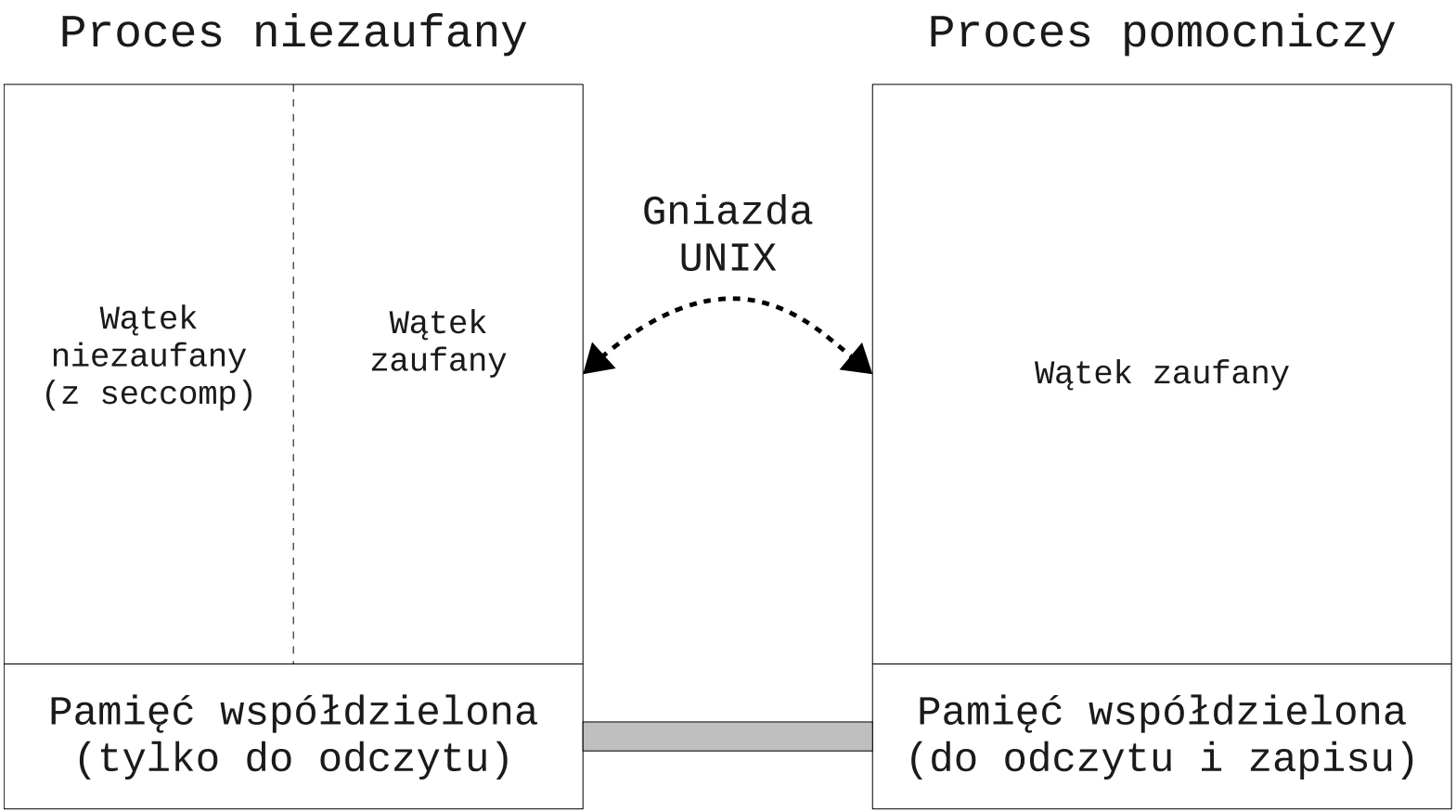
Zalety:

- Dowolna polityka
- Praktycznie zerowy narzut (zależy od polityki)

Wady:

- Możliwość bardzo łatwego ominięcia
- Tylko dla procesów linkowanych dynamicznie
- Brak możliwości modyfikacji w czasie działania

Chromium sandbox, seccomp nurse 1/2



Chromium sandbox, seccomp nurse 2/2

- Stworzony dla projektu *Chrome*
- *seccomp nurse* — adaptacja „dla wszystkich”

Zalety:

- Dobry poziom bezpieczeństwa
- Szerokie możliwości definiowania polityki

Wady:

- Koszmarnie wolny
- Brak obsługi wielu wywołań
  np. `dlopen()`, `clone()`, `exec()`

Patch out

- Usuwanie kodu wywołań z jądra
- Trwałe wyłączanie wywołań niebezpiecznych np. `ptrace()`, `vmsplice()`
- Technika używana w produkcji! (Chrome OS)

Zalety:

- Bezpieczeństwo absolutne!

Wady:

- Kompletny brak elastyczności
- Kłopotliwe w zastosowaniu

syscalls cgroup 1/4

- Ja jestem autorem :)
- Wywołania jako zasób (stąd *cgroups*)
- Implementacja w całości w przestrzeni jądra
- `github.com/luksow/syscalls-cgroup`

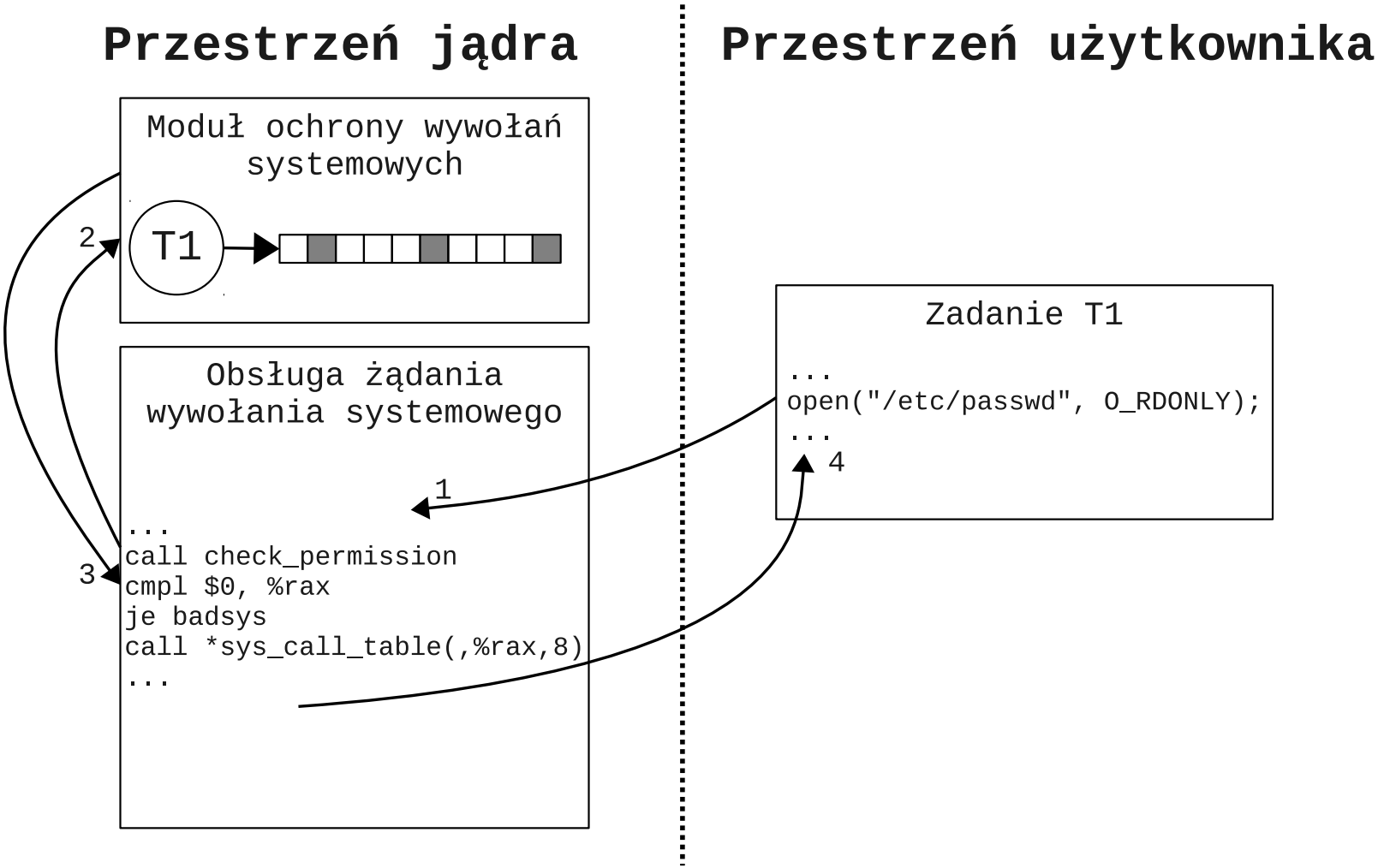
Zalety:

- Wysoka wydajność
- Łatwe w użyciu
- Wygodne w użyciu z kontenerami

Wady:

- Filtrowanie tylko po numerach
- Nie jest w głównej linii jądra

syscalls cgroup 2/4



syscalls cgroup 3/4

```
# mkdir /cgroup
# mount -t tmpfs cgroup_root /cgroup
# cd /cgroup
# mkdir syscalls
# mount -t cgroup -o syscalls syscalls_root syscalls
# ls
cgroup.clone_children cgroup.procs release_agent syscalls.deny
cgroup.event_control notify_on_release syscalls.allow tasks
# cat syscalls.allow
0 1 ... 311
# cat syscalls.deny

# cat tasks
1
2
...
# mkdir test1
# cd test1
# ls
cgroup.clone_children cgroup.procs syscalls.allow tasks
cgroup.event_control notify_on_release syscalls.deny
# echo 0 > tasks
# cat tasks
2357
2374
```

syscalls cgroup 4/4

```
# echo 83 > syscalls.deny # assume 83 is syscall number for 'mkdir'
# cat syscalls.deny
83
# mkdir test2
mkdir: cannot create directory 'test2': Function not implemented
# echo 83 > syscalls.allow
# mkdir test2
# cd ..
# echo 83 > syscalls.deny
# cd test1
# mkdir test3
mkdir: cannot create directory 'test3': Function not implemented
# echo 83 > syscalls.allow
-bash: echo: write error: Operation not permitted
```

seccomp, seccomp filters 1/6

- Rozwiązanie nowe (od 3.5) i wiodące
- Zbudowane na bazie seccomp mode 1
- Używane m. in. w: *OpenSSH*, *vsftpd*, *systemd*

Zalety:

- Wysokie bezpieczeństwo
- Filtrowanie po numerach i argumentach
- Możliwość konfiguracji efektów

Wady:

- Wydajność „tylko” akceptowalna
- Obskurna konfiguracja za pomocą *BPF*

seccomp, seccomp filters 2/6

```
struct sock_filter filter[] = {
  /* Grab the system call number */
  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_nr),
  /* Jump table for the allowed syscalls */
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_rt_sigreturn, 0, 1),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_sigreturn, 0, 1),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_exit_group, 0, 1),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_exit, 0, 1),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_read, 1, 0),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_write, 3, 2),

  /* Check that read is only using stdin. */
  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_arg(0)),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDIN_FILENO, 4, 0),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),

  /* Check that write is only using stdout */
  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_arg(0)),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDOUT_FILENO, 1, 0),
  /* Trap attempts to write to stderr */
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDERR_FILENO, 1, 2),

  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRAP),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
};
```

seccomp, seccomp filters 3/6

```
struct sock_filter filter[] = {
  LOAD_SYSCALL_NR,
  SYSCALL(__NR_exit, ALLOW),
  SYSCALL(__NR_exit_group, ALLOW),
  SYSCALL(__NR_write, JUMP(&l, write_fd)),
  SYSCALL(__NR_read, JUMP(&l, read)),
  DENY,  /* Don't passthrough into a label */

  LABEL(&l, read),
  ARG(0),
  JNE(STDIN_FILENO, DENY),
  ARG(1),
  JNE((unsigned long)buf, DENY),
  ARG(2),
  JGE(sizeof(buf), DENY),
  ALLOW,

  LABEL(&l, write_fd),
  ARG(0),
  JEQ(STDOUT_FILENO, JUMP(&l, write_buf)),
  JEQ(STDERR_FILENO, JUMP(&l, write_buf)),
  DENY,
  ...
};
```

seccomp, seccomp filters 4/6

```
int main()
{
  struct sock_fprog prog = {
    .filter = filter,
    .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
  };
  bpf_resolve_jumps(&l, filter, sizeof(filter)/sizeof(*filter));

  prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
  prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog);

  /* in seccomp filter, yay! */
  ...
}
```

seccomp, seccomp filters 5/6

Na szczęście jest coraz lepiej:

- systemd

```
[Service]
ExecStart=/bin/echo "I am in a sandbox"
SystemCallFilter=brk mmap access open fstat close read
                 fstat mprotect arch_prctl munmap write
```

- libseccomp

```
seccomp_rule_add(SCMP_ACT_ALLOW, SCMP_SYS(read), 1,
                 SCMP_A0(SCMP_CMP_EQ, fileno(read_stream)));
seccomp_rule_add(SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
                 SCMP_A0(SCMP_CMP_EQ, STDOUT_FILENO));
```

- ...

seccomp, seccomp filters 6/6

Zupełnie nienaukowy test wydajnościowy:

x	`getuid()`	`setuid()`	`chdir()`	`read()`
seccomp	202.52%	38.07%	26.96%	3.47%
syscalls cgroup	11.06%	1.78%	2.81%	0.27%

Metoda:

- 10^8 wywołań
- Prosta polityka zezwalająca
- Narzut $= \frac{t_m - t_r}{t_r} \cdot 100\%$

Do zapamiętania

Filtrowanie wywołań systemowych:

- To mechanizm bezpieczeństwa
- Pozwala ściśle kontrolować możliwości procesu
- Ma wiele potencjalnych zastosowań
- Trudny do implementacji
- Trudny do właściwego zastosowania
- Pomimo tego — szeroko wdrażany

Polecane rozwiązania:

- ***seccomp filters*** — zwykle
- *syscalls cgroup* — gdy wydajność jest krytyczna
- Własne — tylko w szczególnych wypadkach

Koniec

Dziękuję

Pytania, uwagi?