

SCR LISTA4 ROZSZERZONE

Denis Firat

February 2021

Zadanie 5

Zadaniem polecenia strace jest śledzenie interakcji wywołanego programu z systemem, obserwuje on wywołania systemowe oraz sygnały procesu. Polecenie strace, działa tak długo jak wywołany program żyje. Strace przydaje się do diagnozowania pracy programu, możemy diagnozować program poprzez uruchomienie programu z strace lub "podpiąć" się do już istniejącego procesu. Chyba najważniejsza opcja jest -e, gdzie dobierając wartość możemy decydować o tym, jakie informacje śledzimy. W dokumentacji strace mamy wypisane możliwe informacje do śledzenia. Kolejna ciekawa opcja to -p, która pozwala nam przyczepić się do już istniejącego procesu i go obserwować. Dokumentacja polecenia strace jest obszerna, co tylko dowodzi jego popularności przy diagnozie programów.

Zadanie 6

Do tego zadania wykorzystałem program zadanie1.c, zakomentowałem tylko część programu odpowiedzialną za nieskończone zapętlenie. Gdy przeprowadzę doświadczenia z poleceniem strace na uproszczonej wersji, sprawdzę jak zachowuje się strace w bardziej "ekstremalnych" warunkach. Po uruchomieniu polecenia strace z programem, można zauważyć całą serię wywołań systemowych, a dopiero na samym końcu jest polecenie write("Witam pozdrawiam").

```
#nload -e /mnt/c:/Users/Denis/Firat/documents/github/src-laboratoria/Lab9$ strace ./a.out
execve("/usr/bin/a.out", {"/a.out"}, 0x7ffffef6e800 / 21 vars...) = 0
open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 0
access("/etc/lid.so.preload", R_OK) = -1 ENOENT (No such file or directory)
mmap(AT_FDCWD, "/etc/lid.so.cache", 0_RDONLY|O_CLOEXEC) = 3
read(3, "\x0d\x05\xfd\x00\xff", 4096) = 4096
mmap(NULL, 38166, PROT_READ, MAP_PRIVATE, 3, 0x7f99f85b7800) = 0
close(3)
open(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", 0_RDONLY|O_CLOEXEC) = 3
read(3, "\x17FEL2\i\j\k\l\m\n\o\p\q\r\s\t\u\v\w\x\xy\z{\}|~\x00\x01\x02...\x82", 832) = 832
pread64(3, "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f...", 784, 64) = 784
pread64(3, "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f...", 832, 848) = 832
pread64(3, "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f...", 896, 896) = 896
stat(3, {\st_mode=S_IFREG|0755, st_size=289224, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f99f85b8000
pread64(3, "\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f...", 784, 64) = 784
pread64(3, "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f...", 832, 896) = 896
pread64(3, "\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f...", 68, 880) = 68
mmap(NULL, 263696, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0x7f99f85b8000)
protect(3, 0x7f99f85b8000, 1287296, PROT_NONE) = 0
mmap(3, 263696, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25800) = 0x7f99f856a000
mmap(3, 0x7f99f856a000, 813194, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d6000) = 0x7f99f851d000
mmap(3, 0x7f99f851d000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f99f8568000
mmap(3, 0x7f99f8568000, 13258, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f99f8568e00
arch_prlctl(ARCH_SET_FS, 0x7f99f85b1340) = 0
protect(3, 0x7f99f8568e00, 12288, PROT_NONE) = 0
```

Figure 1: Odpowiedź polecenia strace przy uproszczonym programie

Najczęściej występujące wywołania: `pread64`, `mmap`, `archprctl` oraz `brk`. Szczerze powiedziawszy, ciężko mi zinterpretować wyniki polecenia `strace`, `mmap` służy do odwzorowywania części pliku w przestrzeni adresowej, `pread64` czytuje bity z deskryptora pliku, a `brk` zmienia rozmiar segmentacji danych. Tak jak mniej więcej rozumiem poszczególne wywołania, tak ciężko mi prześledzić co pokolei się dzieje podczas uruchomienia naszego programu. Ciekawiej zrobiło się, gdy uruchomiłem program `zadanie1.c` z odkomentowanym blokiem nieskończonego zapętlenia. Sytuacja hipotetyczna (głównie dlatego, że kompiler ostrzegł mnie przed zapętleniem funkcji `sleep` oraz dlatego, że wdrożyłem to świadomie), ale założmy, że nie zdawałbym sobie sprawy z nieskończonej petli. Uruchomiłbym program i czekał na jego zakończenie i jak łatwo się domyśleć, nie doczekałbym się.

```
mprotect(0x7f545a89f000, 4096, PROT_READ) = 0
mprotect(0x7f545a8bd000, 4096, PROT_READ) = 0
munmap(0x7f545a894000, 30166) = 0
fstat(1, {st_mode=S_IFCHR|0660, st_rdev=makedev(0x4, 0x1), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL) = 0x7ffffcf2b000
brk(0x7ffffcf2d7000) = 0x7ffffcf2d7000
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, *X0x7ffffd7bdb7f0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, ^Cstrace: Process 62 detached
^detached ...)
```

Figure 2: Odpowiedź polecenia strace przy nieskończonej petli

Na pomoc przychodzi polecenie `strace`, które w jasny sposób wskazuje mi źródło problemu czyli ciągle wywoływanie `clocknanosleep()`, teraz wystarczy wyszukać w programie miejsca gdzie wywołuje funkcje `sleep` i znaleźć winowajcę.

Zadanie 7

Program z listy zkompiłował się bez problemu, po uruchomieniu wyrzuca dobre mi znany błąd segmentation fault, już na tym etapie jest to dla mnie wiadomość, że prawdopodobnie gdzieś wychodzimy poza zakres przydzielonej pamięci (często zdarza się ten błąd przy pracy na tablicach).

[illegible]

Figure 3: Odpowiedź strace na program zadanie7.c

Z pomocą polecenia `strace` uruchomiłem program `zadanie7.c`, po wstępnej obserwacji zauważyłem, że coś jest nie tak przy wywoływaniu `write()`. Uruchomiłem jeszcze raz program z użyciem polecenia `strace`, tym razem użyłem opcji `-e write`, która pozwoliła mi odsiać nie interesujące mnie wywołania. Wywołanie `write()` działało w porządku, do momentu gdy kończyły się trzy kropki w "Witajcie moi mili ..." ponieważ zaraz potem pojawiały się dziwne znaki. Po sprawdzeniu jak w programie wyświetlany jest napis wszystko stało się jasne. Brak ograniczenia na petli `for` sprawił, że znaki z tablicy znaków, przekazywane były nawet po przekroczeniu zakresu tablicy co powodowało pojawianie się tych znaków. Dopiero ograniczenie wykonania petli do np. 5 sprawiło, że program zadziałał poprawnie.