# Программирование в Linux

Межпроцессное взаимодействие.
ptrace

```c
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Системный вызов, позволяющий контролировать исполнение других процессов (тредов).

Основное назначение — отладка

- Смотреть/изменять значения регистров (GETREGS/SETREGS, PEEKUSER/POKEUSER)
- Смотреть/изменять значения в памяти (PEEKTEXT/POKETEXT, PEEKDATA/POKEDATA)
- Отслеживать получаемые сигналы (GETSIGINFO/SETSEGINFO)
- Контролировать блокировку сигналов (GETSIGMASK/SETSIGMASK)
- Перехватывать системные вызовы (SYSCALL)
- Выполнять по шагам (SINGLESTEP)

# Подключаемся

ptrace(PTRACE_TRACEME, 0, NULL, NULL)

Этот вызов выполняет дочерний процесс, подключая родительский в качестве отладчика. Дочерний процесс не останавливается

Можно проверить, не запустили ли нас под ptrace, или избежать отладки

```
7   int main() {
8       errno = 0;
9       ptrace(PTRACE_TRACEME, 0, nullptr, nullptr);
10      if (errno) {
11          std::cout << "DO NOT PTRACE ME!!!\n";
12      } else {
13          std::cout << "Hello world!\n";
14      }
15  }
```

```
dmis@dmis-MS-7A15:~/LinuxEgs/ptrace_egs$ ./refuse
Hello world!
dmis@dmis-MS-7A15:~/LinuxEgs/ptrace_egs$ gdb -q ./refuse
Reading symbols from ./refuse...
(No debugging symbols found in ./refuse)
(gdb) run
Starting program: /home/dmis/LinuxEgs/ptrace_egs/refuse
DO NOT PTRACE ME!!!
```

```
dmis@dmis-MS-7A15:~$ gdb -q --pid $(pidof refuse)
Attaching to process 2670720
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 2670720 is already traced by process 2666193
ptrace: Operation not permitted.
```

# Подключаемся

ptrace(PTRACE_ATTACH, pid, NULL, NULL)

Пытаемся подключиться к отладке существующего процесса/треда pid.

Отлаживаемый процесс получит SIGSTOP.

Отладчику стоит дождаться этого момента (waitpid)

ptrace(PTRACE_SEIZE, pid, NULL, PTRACE_O_flags)

Пытаемся подключиться к отладке существующего процесса/треда pid.

Отлаживаемый процесс не будет остановлен

Нужно либо использовать TRACEME, либо ATTACH/SEIZE.
Отлаживать самого себя нельзя.
Но можно сделать цикл из отладчиков.

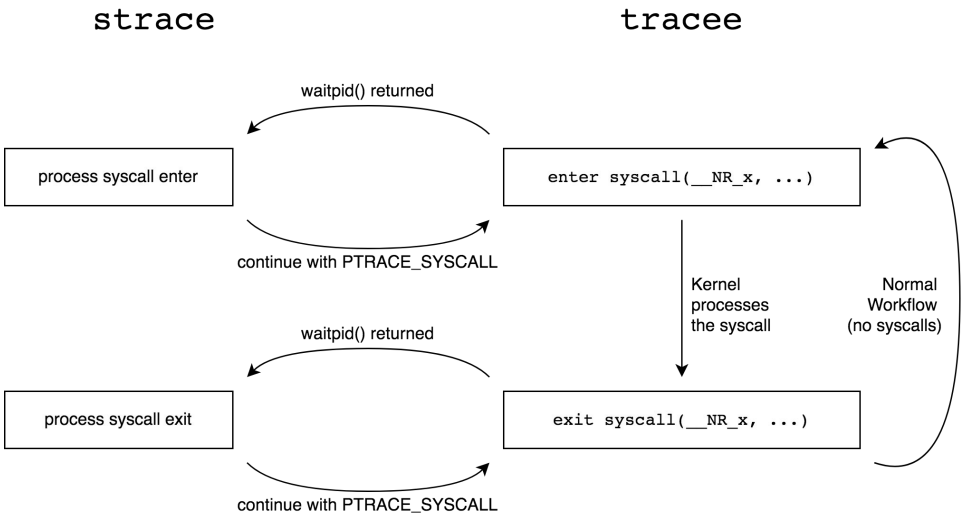
# Как все взаимодействует?

Отлаживаемый процесс начинает останавливаться при получении сигналов (обработка не начинается)

Процесс-отладчик уведомляется о пришедшем сигнале (и может его перехватить и отменить) и об остановке/продолжении процесса (SIGCHLD)

Отлаживаемый процесс получает SIGTRAP при наступлении отслеживаемых событий

SIGTRAP может генериться, например, прерыванием №3 (однобайтовая инструкция — помещается в начало любой инструкции в секции кода как breakpoint)



```
21  static void tracee(char **argv)
22  {
23      if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
24          die("child: ptrace(traceme) failed: %m");
25
26      /* Остановиться и дождаться, пока отладчик отреагирует. */
27      if (raise(SIGSTOP))
28          die("child: raise(SIGSTOP) failed: %m");
29
30      /* Запустить процесс. */
31      execvp(argv[0], argv);
32
33      /* Сюда выполнение дойти не должно. */
34      die("tracee start failed: %m");
35  }
36
37  static void tracer(pid_t pid)
38  {
39      int status = 0;
40
41      /* Дождаться, пока дочерний процесс сделает нас своим отладчиком. */
42      if (waitpid(pid, &status, 0) < 0)
43          die("waitpid failed: %m");
44      if (!WIFSTOPPED(status) || WSTOPSIG(status) != SIGSTOP) {
45          kill(pid, SIGKILL);
46          die("tracer: unexpected wait status: %x", status);
47      }
48      /* Если требуются дополнительные опции для ptrace, их можно задать здесь. */
49      /* Начиная с этого момента можно использовать PTRACE_SYSCALL. */
50
51      sleep(5);
52  }
```

```
55  void test_ptrace(char **argv)
56  {
57      pid_t pid = fork();
58      if (pid < 0)
59          die("couldn't fork: %m");
60      else if (pid == 0)
61          tracee(argv);
62      else
63          tracer(pid);
64  }
65
66  int main(int argc, char* argv[]) {
67      if (argc == 1) {
68          return EXIT_FAILURE;
69      }
70      std::vector<char*> args;
71      for (int i = 1; i < argc; ++i) {
72          args.push_back(argv[i]);
73      }
74      args.push_back(nullptr);
75      test_ptrace(args.data());
76  }
```

# PTRACE_SYSCALL

ptrace(PTRACE_SYSCALL, pid, NULL, NULL)
Продолжает выполнение процесса до входа или до выхода из системного вызова.

Отладчик должен помнить, где он находится: на входе или на выходе

Номер выполняемого системного вызова передается платформоспецифичным способом. Под x86 — регистр rax

```
for (;;) {
    /* Enter next system call */
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, 0, 0);

    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, 0, &regs);

    /* Is this system call permitted? */
    int blocked = 0;
    if (is_syscall_blocked(regs.orig_rax)) {
        blocked = 1;
        regs.orig_rax = -1; // set to invalid syscall
        ptrace(PTRACE_SETREGS, pid, 0, &regs);
    }

    /* Run system call and stop on exit */
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, 0, 0);

    if (blocked) {
        /* errno = EPERM */
        regs.rax = -EPERM; // Operation not permitted
        ptrace(PTRACE_SETREGS, pid, 0, &regs);
    }
}
```

# Лезем в адресное пространство

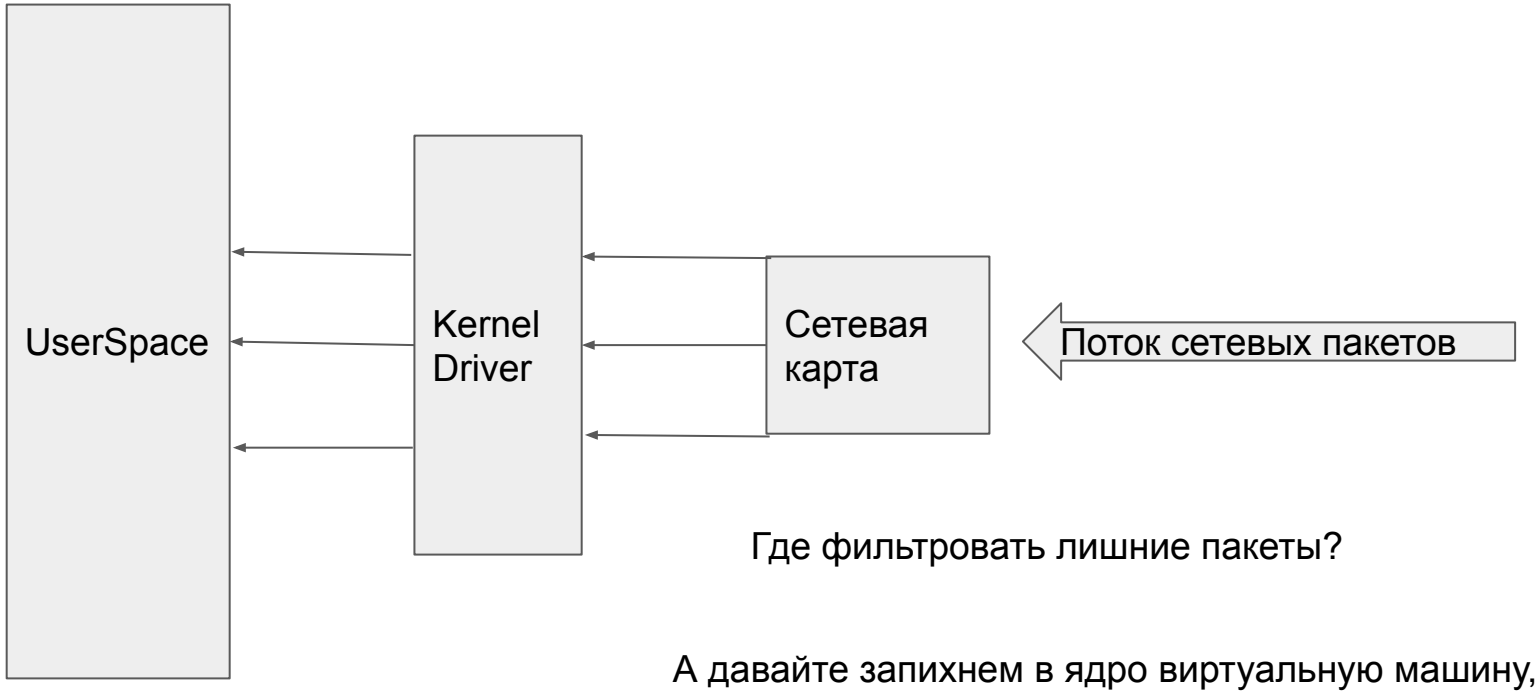
ptrace позволяет писать/читать адресное пространство по одному машинному слову (long)

Более удобно:

```c
#include <sys/uio.h>

ssize_t process_vm_readv(pid_t pid,
                         const struct iovec *local_iov,
                         unsigned long liovcnt,
                         const struct iovec *remote_iov,
                         unsigned long riovcnt,
                         unsigned long flags);

ssize_t process_vm_writev(pid_t pid,
                          const struct iovec *local_iov,
                          unsigned long liovcnt,
                          const struct iovec *remote_iov,
                          unsigned long riovcnt,
                          unsigned long flags);
```

```cpp
static void handle_syscall(pid_t pid, const user_regs_struct& regs) {
    if (regs.orig_rax != SYS_write) {
        return;
    }
    // rdi -- fd
    // rsi -- buf ptr
    // rdx -- len
    printf("SYS_write: base=%llx len=%lld\n", regs.rsi, regs.rdx);

    iovec tracee_bufer {
        .iov_base = reinterpret_cast<void*>(regs.rsi),
        .iov_len  = regs.rdx
    };
    std::vector<char> buffer(tracee_bufer.iov_len + 1, 0);
    iovec local_buffer {
        .iov_base = buffer.data(),
        .iov_len  = tracee_bufer.iov_len
    };
    auto sz = process_vm_readv(pid,
                               &local_buffer, 1,
                               &tracee_bufer, 1,
                               0);
    if (sz < 0) {
        perror("process_vm_readv");
    }
    if (sz == tracee_bufer.iov_len) {
        printf("Successfully read\n");
    }
    printf("Tracee invoked SYS_write with data: %s\n", buffer.data());
    fflush(stdout);

    std::fill_n(buffer.data(), tracee_bufer.iov_len - 1, 'A');
    buffer[tracee_bufer.iov_len - 1] = '\n';
    sz = process_vm_writev(pid,
                           &local_buffer, 1,
                           &tracee_bufer, 1,
                           0);
    if (sz < 0) {
        perror("process_vm_writev");
        printf("Cannot write tracee\n");
    }
}
```

# BPF (Berkeley Packet Filters)




Где фильтровать лишние пакеты?

А давайте запихнем в ядро виртуальную машину, которая будет крутить пользовательские фильтры пакетов!

# Старый классический BPF

**Виртуальная машина (RISC-архитектура)**
2 32-битных регистра:
A — аккумулятор
X — индекс

64 байта памяти (сюда размещались байты пакета)

В инструкциях для машины можно использовать условные переходы, но запрещены циклы (для гарантии завершимости)

**Схема использования**

Написать программу фильтра.
Сгенерировать байткод
Загрузить в ядро
Подключить загруженный фильтр к подсистеме, генерирующей события

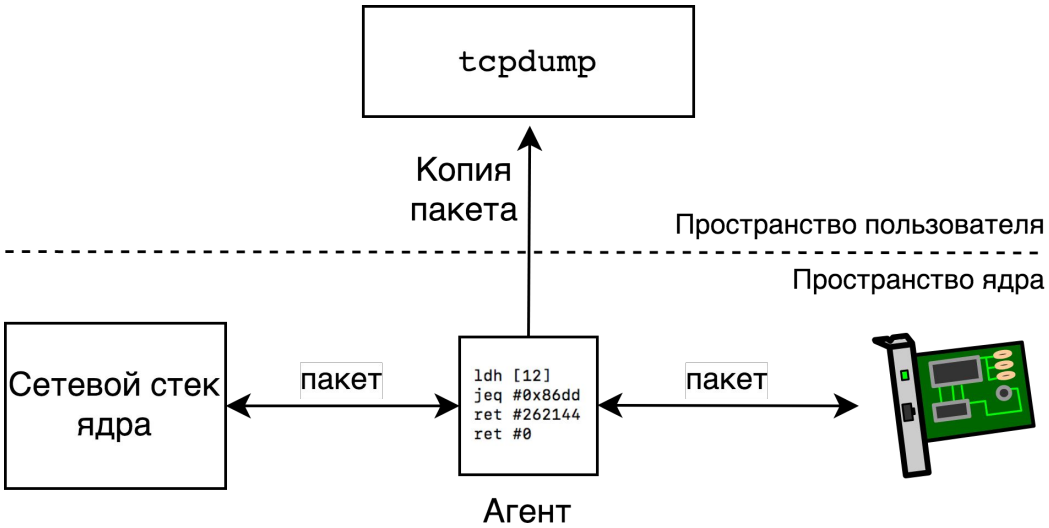
## Пример

```
sudo tcpdump -i enp2s0 ip6 # слушаем только ipv6 пакеты
```

```
dmis@dmis-MS-7A15:~$ sudo tcpdump -i enp2s0 -d ip6
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

6 6 2

|Destination MAC|Source MAC|Ether Type|...|




```cpp
#include <algorithm>

#include <linux/filter.h>
#include <linux/if_ether.h>

#include <sys/types.h>
#include <sys/socket.h>

void SetIPV6Filter(int socket_fd) {
    sock_filter code[] = {
        BPF_STMT(BPF_LD|BPF_H|BPF_ABS, 12),
        BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, ETH_P_IPV6, 0, 1),
        BPF_STMT(BPF_RET|BPF_K, 0x00040000),
        BPF_STMT(BPF_RET|BPF_K, 0),
    };
    sock_fprog prog = {
        .len = std::size(code),
        .filter = code,
    };
    setsockopt(socket_fd,
               SOL_SOCKET, // применить на уровне сокета
               SO_ATTACH_FILTER,
               &prog,
               sizeof(prog));
}
```

# seccomp* — применяем BPF к контролю syscalls

```
#include <linux/seccomp.h>
#include <linux/filter.h>
#include <linux/audit.h>
#include <linux/signal.h>
#include <sys/ptrace.h>

int seccomp(unsigned int operation, unsigned int flags, void *args);
```

seccomp(SECCOMP_SET_MODE_FILTER, flags, &filter)

В случае системных вызовов, в память VM ложится структура:

```
struct seccomp_data {
    int   nr;                   /* System call number */
    __u32 arch;                 /* AUDIT_ARCH_* value
                                   (see <linux/audit.h>) */
    __u64 instruction_pointer;  /* CPU instruction pointer */
    __u64 args[6];              /* Up to 6 system call arguments */
};
```

пример:

```
ld [0]
jeq #304, bad
jeq #176, bad
jeq #239, bad
jeq #279, bad
good: ret #0x7fff0000 /* SECCOMP_RET_ALLOW */
bad: ret #0
```

Проверить код фильтра: [seccomp_check_filter()]

В функции багофича: инструкцию mod нельзя использовать в фильтрах для seccomp

* справедливо для версии 2012 года и новее

# libseccomp

```c
#include <seccomp.h>
#include <unistd.h>
#include <errno.h>
#include <err.h>

static int sys_numbers[] = {...

int main(int argc, char **argv)
{
    scmp_filter_ctx ctx;
    size_t i;

    if (argc < 2) {
        fprintf(stderr, "usage: seccomp_lib <prog>\n");
        exit(1);
    }

    ctx = seccomp_init(SCMP_ACT_ALLOW);
    if (!ctx)
        err(1, "seccomp_init");

    for (i = 0; i < sizeof(sys_numbers)/sizeof(sys_numbers[0]); i++)
        seccomp_rule_add(ctx, SCMP_ACT_TRAP, sys_numbers[i], 0);

    seccomp_load(ctx);

    execvp(argv[1], &argv[1]);
    err(1, "execlp: %s", argv[1]);
}
```

- Процесс и все его потомки наследуют установленные параметры seccomp
- Более переносимая обертка над системным вызовом seccomp
- Использование фильтров дает меньший оверхед чем ptrace(PTRACE_SYSCALL)

```
dmis@dmis-MS-7A15:~/LinuxEgs/ptrace_egs$ g++ -std=c++17 -o sec_test seccomp_egs.cpp -lseccomp
dmis@dmis-MS-7A15:~/LinuxEgs/ptrace_egs$ ./sec_test echo "hello"
hello
dmis@dmis-MS-7A15:~/LinuxEgs/ptrace_egs$ sudo ./sec_test mount -t bpf bpf /tmp/
Bad system call
```