# ptrace

this system call is highly used when analyzing, debugging, reverse engineering, and modifying programs that use *ELF* format. *ptrace* allows us to attach to a process and access code, data, stack, heap, and registers.

Once *ELF* is completely mapped, we can attach to process, parse or modify *ELF* similarly as we do in file on disk. *ptrace* gives access the program instead of using *open/mmap/read/write* calls.

*ptrace* gives full control over program's execution flow, this allows memory virus infection, virus analysis or even detection of userland memory rootkits, hotpatching, reverse engineering.

## Importance of ptrace

Someone can attach to a process that they own and modify, analyze, reverse, and debug it. *gdb*, *strace*, and *ltrace* make use of *ptrace*. It gives a programmer ability to attach to a process and modify the memory, which can include injecting and modifying data structures such as **Global Offset Table (GOT)** for shared library redirection.

## ptrace requests

*ptrace* system call has a *libc* wrapper, you may include *ptrace.h* and call *ptrace* while passing it a request and a process ID.

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

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

## Request Types (enum ___ptrace__request)

| Request | Description |
|---|---|
| *PTRACE_ATTACH* | Attach to process specified by *pid*, making it a tracee of calling process. Tracee is sent a *SIGSTOP* signal, but will not have stopped by completion of this call. Use *waitpid* to wait for tracee to stop. |
| *PTRACE_TRACEME* | Indicates this process is to be traced by its parent. A process shouldn't make this if its parent isn't expecting to trace it. |

| Request | Description |
|---|---|
| *PTRACE_PEEKTEXT*<br>*PTRACE_PEEKDATA*<br>*PTRACE_PEEKUSER* | Allow tracing process to read from a virtual memory address within traced process image; we can read entire text or data segment into a buffer for analysis. No difference in implementation between these requests. |
| *PTRACE_POKETEXT*<br>*PTRACE_POKEDATA*<br>*PTRACE_POKEUSER* | Requests allow tracing process to modify any location within traced process image. |
| *PTRACE_GETREGS* | Allows tracing process to get a copy of traced process's registers. Each thread context has its own register set. |
| *PTRACE_SETREGS* | Allows tracing process to set new register values for traced process, allowing modifying for example instruction pointer, to point a shellcode |
| *PTRACE_CONT* | Tells stopped traced process to resume execution |
| *PTRACE_DETACH* | resumes traced process and detaches from it |
| *PTRACE_SYSCALL* | Request resumes traced process but arranges for it to stop at entrance/exit of next syscall. Allows us to inspect arguments for syscall and even modify them. *ptrace* request heavily used in code for a program called *straced*, which traces all system calls when a program runs |
| *PTRACE_SINGLESTEP* | Resumes process but stops it after next instruction. Single stepping allows a debugger to stop after every instruction executed. Allows a user to inspect values of registers and state of process after each instruction |
| *PTRACE_GETSIGINFO* | Retrieves information about signal that caused the stop. It retrieves a copy of *siginfo_t* structure, this can be modified and set (with next request) |
| *PTRACE_SETSIGINFO* | Sets signal information. This will affect only signals that would normally be delivered to tracee and would be caught by the tracer (*addr* ignored) |
| *PTRACE_SETOPTIONS* | sets *ptrace* options from *data* (*addr* ignored). Data is interpreted as a bitmask of options. |

Two terms: * *tracer* process doing the tracing (invoking *ptrace*). * *tracee* or *traced*: program being traced by tracer.

## The process register state and flags

Depending on the architecture we are going to trace we're gonna have different struct of registers, but mainly we will have general-purpose registers, segmentation registers, stack pointer, CPU flags, and TLS registers:

**x86-64**

```c
struct user_regs_struct
{
  __extension__ unsigned long long int r15;
  __extension__ unsigned long long int r14;
  __extension__ unsigned long long int r13;
  __extension__ unsigned long long int r12;
  __extension__ unsigned long long int rbp;
  __extension__ unsigned long long int rbx;
  __extension__ unsigned long long int r11;
  __extension__ unsigned long long int r10;
  __extension__ unsigned long long int r9;
  __extension__ unsigned long long int r8;
  __extension__ unsigned long long int rax;
  __extension__ unsigned long long int rcx;
  __extension__ unsigned long long int rdx;
  __extension__ unsigned long long int rsi;
  __extension__ unsigned long long int rdi;
  __extension__ unsigned long long int orig_rax;
  __extension__ unsigned long long int rip;
  __extension__ unsigned long long int cs;
  __extension__ unsigned long long int eflags;
  __extension__ unsigned long long int rsp;
  __extension__ unsigned long long int ss;
  __extension__ unsigned long long int fs_base;
  __extension__ unsigned long long int gs_base;
  __extension__ unsigned long long int ds;
  __extension__ unsigned long long int es;
  __extension__ unsigned long long int fs;
  __extension__ unsigned long long int gs;
};
```

**x86-32**

```c
struct user_regs_struct
{
  long int ebx;
  long int ecx;
  long int edx;
  long int esi;
  long int edi;
  long int ebp;
  long int eax;
  long int xds;
  long int xes;
  long int xfs;
  long int xgs;
  long int orig_eax;
  long int eip;
  long int xcs;
  long int eflags;
  long int esp;
  long int xss;
};
```

**Notes**

The structure contains an *orig_eax* which contains in a syscall the number of the syscall, this can be used in the entry or the exit of a syscall to know the syscall number. In Linux to search for the **thread-local-storage (TLS)** we must use the register *%gs* in 32 bits, and *%fs* in 64 bits.

## Implementing a simple Debugger

Complete code is in *ptrace_debugger.c*, we'll see here the most important parts of the equation.

This code only works on binaries of 64 bits for Intel, and compiled as executables, so no code in the form of PIC or PIE is found (all the virtual addresses in the segments are absolute addresses, not relative virtual addresses). So the first part of the program just retrieves the program name and a function to set a breakpoint:

```
if ((h.exec = strdup(argv[1])) == NULL)
{
    perror("executable strdup");
    exit(-1);
}
args[0] = h.exec;
args[1] = NULL;

if ((h.symname = strdup(argv[2])) == NULL)
{
    perror("symname strdup");
    exit(-1);
}
```

All the data is stored in a structure developed for the program.

Then, file is read into a buffer and parsed with ELF structures, the only part we'll see from here will be the check done to the binary in order to debug it or not. The first check is done to know if is a 64 bit binary:

```
if (h.ehdr->e_machine != EM_IA_64 && h.ehdr->e_machine != EM_X86_64)
{
    fprintf(stderr, "Only supported x86_64 elf binaries\n");
    exit(-2);
}
```

Second one is to know if binary is an EXE, or other type of binary (it could be a DYN for example). Only those compiled without PIE will be allowed:

```
if (h.ehdr->e_type != ET_EXEC)
{
    fprintf(stderr, "%s is not an ELF executable\n", h.exec);
    exit(-1);
}
```

Finally, we will check for symbol strings, and sections (in order to retrieve symbols), this is done to retrieve the address of the function we gave as parameter:

```
if (h.ehdr->e_shstrndx == 0 || h.ehdr->e_shoff == 0 || h.ehdr->e_shnum == 0)
{
    fprintf(stderr, "Section header table not found\n");
    exit(-1);
}

if ((h.symaddr = lookup_symbol(&h, h.symname)) == 0)
{
    fprintf(stderr, "Unable to find symbol: %s not found in executable\n", h.symname);
    exit(-1);
}
```

**How to implement tracer and tracee**

Two different implementations can be done to create both tracer and tracee, both implementations start by the same code, create a child process, this can be done using the syscall *fork*, syscall creates a child process as a copy of the parent from the beginning of the code to that moment, for child process it returns a value 0, and for parent process return other value greater than 0, which represent the pid of child process.

- Unusual implementation: in this implementation the parent process is the tracee, and the child process is the tracer. Parent process can do the next:

```
prctl(PR_SET_PTRACER, child_pid, 0, 0, 0);
sleep(X);
```

So in this case, the parent process tells specifically that its child process can trace it, and sleep some time until child process attach to it.

The child process, must receive the parent process PID in order to call *PTRACE_ATTACH*.

```
ptrace(PTRACE_ATTACH, parent_process, 0x0, 0x0);
```

This will not be in any case, the way that we will implement it is the next.

- Implementation we will use: in this case, is the most common way for the implementation, the parent process will trace the child process. The child process will execute *ptrace* to indicate that it will be traced by parent process, and parent process will wait for child process.

First of all create the process with *fork*:

```
if ((pid = fork()) < 0)
{
    perror("fork");
    exit(-1);
}
```

Then we will have the check for the child process and the execution of the binary given as argument:

```
if (pid == 0) // child process
{
    if (ptrace(PTRACE_TRACEME, pid, NULL, NULL) < 0) // ptrace own
    // process (will be catched by parent process)
    {
        perror("PTRACE_TRACEME");
        exit(-1);
    }

    execve(h.exec, args, envp); // execute new process,
    // never return (this is like creating a process
    // suspended in windows)
    exit(0);
}
```

That code represent all that child process will do in the binary.

Now parent process will wait for child process, nothing more.

```
wait (&status);
```

As we gave as input one of the functions, we will set a breakpoint on that function, previously we obtained the address of the function through the binary symbols, the process to set a breakpoint is to read the address where we gonna set the breakpoint so we will have the original byte to restore it later, and we will set the trap byte. A software breakpoint is implemented using the byte *0xCC* once the program is gonna execute an instruction with this byte (instruction *int 3*) it will generate an exception, that exception will be catched by parent process (and we should know that it has been generated in the address of the breakpoint).

```
// read address where to write breakpoint
if ((unsigned int)(orig = ptrace(PTRACE_PEEKTEXT, pid,
h.symaddr, NULL)) < 0)
{
    perror("PTRACE_PEEKTEXT");
    exit(-1);
}

// set software interruption
trap = (orig & ~0xff) | 0xcc;

// write instruction with breakpoint
// again into same address
if (ptrace(PTRACE_POKETEXT, pid, h.symaddr, trap) < 0)
{
    perror("PTRACE_POKETEXT");
    exit(-1);
}
```

Once with breakpoint set, and in order to continue process execution we will call again ptrace and we will wait for a new event:

```
if (ptrace(PTRACE_CONT, pid, NULL, NULL) < 0)
{
    perror("PTRACE_CONT");
    exit(-1);
}

wait(&status);
```

The child process will continue, and in the moment that the chosen function is executed, the breakpoint will halt the process and will wake up the parent process from the wait. This will return in the status the reason, we can check the reason with different macros, in here we will use *WIFSTOPPED* to check if child is stopped, and with *WSTOPSIG* we will check the stop status, the reason should be **SIGTRAP**.

```
if (WIFSTOPPED(status) && WSTOPSIG(status) == SIGTRAP)
{
```

We can now get the register values using one of the previous structures depending on the architecture:

```
if (ptrace(PTRACE_GETREGS, pid, NULL, &h.pt_reg) < 0)
```

```c
{
    perror("PTRACE_GETREGS");
    exit(-1);
}
```

Once the breakpoint has halt the program, we have to follow various steps in order to:

- recover the original byte from the instruction.
- fix the program counter to point again to the original instruction.
- execute only that instruction.
- set breakpoint again.
- continue the execution.

```c
// write the original byte again back to the address
if (ptrace(PTRACE_POKETEXT, pid, h.symaddr, orig) < 0)
{
    perror("PTRACE_POKETEXT");
    exit(-1);
}

// fix the program counter, so point again to the instrucction
h.pt_reg.rip = h.pt_reg.rip - 1;

// Set the values back to the process
if (ptrace(PTRACE_SETREGS, pid, NULL, &h.pt_reg) < 0)
{
    perror("PTRACE_SETREGS");
    exit(-1);
}

// Execute only one instrucction (the one fixed)
if (ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL) < 0)
{
    perror("PTRACE_SINGLESTEP");
    exit(-1);
}

// wait for single step
wait(NULL);

// restore software breakpoint byte
if (ptrace(PTRACE_POKETEXT, pid, h.symaddr, trap) < 0)
{
    perror("PTRACE_POKETEXT");
    exit(-1);
}
```

Finally we should jump to the *PTRACE_CONT* *ptrace* call.

## Implementing an attacher debugger

In some cases we do want to attach to an already running process. In that case, we need to get the pid of the process we want to attach to. This time we have the code in *ptrace_attacher.c*. The code is similar to the previous one, but this time we will need code to get the pid, and also to get the path of the binary in order to parse it and detect if it's an EXE file of 64 bits.

So this time, we will have a switch statement in order to detect if the user is giving a pid or exe file (as it can debug both this program), and also to get the function name. To do that, we will use the *getopt* function:

```c
while ((c = getopt(argc, argv, "p:e:f:")) != -1)
{
    switch(c)
    {
    case 'p':
        pid = atoi(optarg);
        h.exec = get_exe_name(pid);
        if (h.exec == NULL)
        {
            printf("Unable to retrieve executable path for pid: %d\n", pid);
            exit(-1);
        }
        mode = PID_MODE;
        break;
    case 'e':
        if ((h.exec = strdup(optarg)) == NULL)
        {
            perror("strdup");
            exit(-1);
        }

        mode = EXE_MODE;
        break;
    case 'f':
        if ((h.symname = strdup(optarg)) == NULL)
        {
            perror("strdup");
            exit(-1);
        }
        break;
    default:
        printf("Unknown option\n");
        break;
    }
}
```

If pid is given instead of exe, the next code is executed in order to retrieve the exe path from the command line. This is possible in Linux thanks to the path */proc//cmdline*, this would be the code of *get_exe_name*:

```
char*
get_exe_name(int pid)
{
    char cmdline[255], path[512], *p;
    int fd;
    snprintf(cmdline, 255, "/proc/%d/cmdline", pid);

    if ((fd = open(cmdline, O_RDONLY)) < 0)
    {
        fprintf(stderr, "Error opening file %s\n", cmdline);
        perror("open");
        exit(-1);
    }

    if (read(fd, path, 512) < 0)
    {
        perror("read");
        exit(-1);
    }

    if ((p = strdup(path)) == NULL)
    {
        perror("strdup");
        exit(-1);
    }

    return p;
}
```

As we will attach to the program, we will set a handler for keyboard codes as *CTRL+C*, this handler is set with a call to the function **signal** giving as first parameter the signal to handle (in this case **SIGINT**) and as second parameter the function. In this function what we would do is to *dettach* from the traced process:

```
void
sighandler(int sig)
{
    printf("Caught SIGINT: Detaching from %d\n", global_pid);
    if (ptrace(PTRACE_DETACH, global_pid, NULL, NULL) < 0 && errno)
    {
        perror("PTRACE_DETACH");
        exit(-1);
    }

    exit(0);
}
```

Once we obtain the path to the binary, the process is pretty similar to the previous one, we have to parse the binary, and obtain the address of the function from the symbol table of the binary, it could be that no symbol table is present, so execution could not be possible as in this case we rely on the symbols.

Next step will be easier than in previous case, so in previous debugger we had to create another process to trace it, this time we have to attach to the remote process by pid:

```
// attach to process 'pid'
if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0)
{
    perror("PTRACE_ATTACH");
    exit(-1);
}
// finally wait for the other process to stop
wait(&status);
```

The next steps would be exactly the same than in the debugger, as the only part a little bit different is the use of **PTRACE_ATTACH** instead of using the **PTRACE_TRACEME** in the child process.

## Useful Functions

### Read Function

Due to the fact that the **ptrace** call only reads in words with **PTRACE_PEEKDATA**, so in 32 bits we should for example read in blocks of 4 bytes, in case that we need to read an unaligned size of bytes, we can use the next function:

```
long * read_data_from_memory(pid_t pid,long *addr,long *buffer,size_t size)
{
  size_t size_cpy;
  long read_value;
  uint last_value;
  long remainder;
  int i;
  uint mask_byte;

  size_cpy = size;
  if ((int)size < 0) {
    size_cpy = size + 3;
  }
  i = 0;
  while (i < (int)size_cpy >> 2) {
    // read from remote process
    read_value = ptrace(PTRACE_PEEKDATA,pid,addr,0x0);
    *buffer = read_value;
    addr = addr + 1;
    buffer = buffer + 1;
    i = i + 1;
  }

  /* If size is not aligned to long (4)
    read the last part as byte, 2 bytes
    or 3 bytes */

  remainder = size % sizeof(long);

  if (remainder == 0x1) {
    mask_byte = 0xff;
  }
  if (remainder == 0x2) {
    mask_byte = 0xffff;
  }
  if (remainder == 0x3) {
    mask_byte = 0xffffff;
  }
  if (remainder != 0x0) {
    last_value = ptrace(PTRACE_PEEKDATA,pid,addr, 0x0);
    *buffer = *buffer & ~mask_byte | last_value & mask_byte;
  }

  return remainder;
}
```

### Write Function

The same happen with the Write function so we have to go in rounds of 4 bytes (size of long), instead of using **PTRACE_PEEKDATA** to retrieve data (it will be used at the end), we will use **PTRACE_POKEDATA**:

```
void write_in_memory(pid_t current_pid,long *address,void **buffer,size_t size)
{
  long last_value;
  size_t size_cpy;
  int remainder;
  int i;
  uint mask;

  size_cpy = size;
  if ((int)size < 0) {
    size_cpy = size + 3;
  }
  i = 0;
  while (i < (int)size_cpy >> 2) {
    ptrace(PTRACE_POKEDATA,current_pid,address,*buffer);
    address = address + 1;
    buffer = buffer + 1;
    i = i + 1;
  }


  remainder = size % sizeof(long);

  if (remainder == 1) {
    mask = 0xff;
  }
  if (remainder == 2) {
    mask = 0xffff;
  }
  if (remainder == 3) {
    mask = 0xffffff;
  }

  if (remainder != 0) {
    last_value = ptrace(PTRACE_PEEKDATA,current_pid,address, 0x0);
    ptrace(PTRACE_POKEDATA,current_pid,address,
                     (void *)((uint)*buffer & mask | ~mask & last_value));
  }

  return;
}
```

### Avoid Tracing From Other Programs

In the same way that **ptrace** can be used to trace/debug a program and get its state while running, it's possible to trick **ptrace** to avoid other processes to execute ptrace on a given process. So if we call **ptrace** with **PTRACE_ATTACH** giving as pid our own pid, we will set our own process as the tracer.

```
current_pid = getpid();

ptrace_output = ptrace(PTRACE_ATTACH, current_pid, 0x0, 0x0);
```

```c
if (ptrace_output == -1)
{
    puts("Detected a tracer already attached, killing process\n");
    kill(current_pid, 9);
}
```

Also, in case we want to detect if a tracer is already attached to us, we can get its pid from one of the files in */proc//status*, checking line by line, we will find one that start by *"TracerPid"*, reading that line and extracting the value we will get if there's no process tracing us (0) or if there's one process tracing us (pid of tracert):

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MAX_LEN 256

int
GetTracerPidValue(pid_t current_pid)
{
    const char *proc_status = "/proc/%d/status";
    const char *tracerpid = "TracerPid:";
    const char *tracerpid_value = "TracerPid: %d";

    char proc_pid_status[MAX_LEN];
    FILE *fd;
    char *line_buf;
    size_t line_buf_size;
    ssize_t line_size;
    int tracerpid_found;
    size_t tracerpid_len;
    int tracerpid_value_int;

    memset(proc_pid_status, 0, MAX_LEN);

    snprintf(proc_pid_status, MAX_LEN, proc_status, current_pid);
    fd = fopen(proc_pid_status, "r");

    line_buf = NULL;
    line_buf_size = 0;
    tracerpid_found = 0;

    do
    {
        line_size = getline(&line_buf, &line_buf_size, fd);
        if (line_size < 0)
        {
            free(line_buf);
            fclose(fd);
            return 0;
        }
        tracerpid_len = strlen(tracerpid);
        tracerpid_found = memcmp(tracerpid, line_buf, tracerpid_len);
    } while (tracerpid_found != 0);

    sscanf(line_buf, tracerpid_value, &tracerpid_value_int);
    free(line_buf);
    fclose(fd);

    return tracerpid_value_int;
}


int
main()
{
    int tracerpid = GetTracerPidValue(getpid());

    if (tracerpid != 0)
    {
        printf("Debugger detected, debugger's pid %d\n", tracerpid);
        return -1;
    }
    else
        printf("No debugger detected, all fine\n");

    return 0;
}
```

### Managing Syscalls

Whenever you want to manage the syscalls in a program you're tracing, we saw that it's possible to stop in these syscalls using the flag **PTRACE_SYSCALL** in the **ptrace** function, this works similar to **PTRACE_CONT** but when a syscall is executed this will stop at the beginning of the syscall, and with another **PTRACE_SYSCALL** the program will stop at the end of the syscall.

Due to the fact that when a syscall is executed, the registers are modified, we need some way in order to know which syscall is going to be executed, for that reason in the structures of registers, we have an specific one with this number, if we know from assembly programming, the syscall number is set in *eax* or *rax* depending on the architecture, well, in the structures we have a field like *orig_eax* or *orig_rax*, these value holds the syscall number. Finally, once a syscall has finished we will have the return value in *eax* or *rax*.

To handle the syscall, it will be the same method than handling a breakpoint, we will have a **SIGSTOP**, and the reason will be a **SIGTRAP**. If we have a list of breakpoints, we should retrieve the value from the program counter (*eip* or *rip*) in order to check if that value - 1 (because the program counter holds the next instruction) is one of our breakpoint addresses. Also other way would be to read the program counter value - 1 memory and check if it contains a *0xCC* (software breakpoint), in that case handle the breakpoint as we saw previously and nothing more.

In order to know if we are at the beginning of a syscall or at the end, we can just use a "boolean" value with 1 and 0, these would be an example:

```
if (WIFSTOPPED(status) && WSTOPSIG(status) == SIGTRAP)
{
    // get registers from program
    if (ptrace(PTRACE_GETREGS, pid, NULL, &h.pt_reg) < 0)
    {
        perror("PTRACE_GETREGS");
        exit(-1);
    }

    if (h.pt_reg.rip != (h.symaddr+1))
    {
        if (is_entry)
        {
            printf("Stop in a syscall, number of syscall: %d\n", h.pt_reg.orig_rax);
            is_entry = 0;
        }
        else
        {
            printf("End of syscall, returns: %016x\n", h.pt_reg.rax);
            is_entry = 1;
        }
    }
```

If you remember in *h.symaddr* we had the address of the function where we set the breakpoint, this is just a simple way to manage the syscall.

### How to avoid the execution of a syscall

As written by nullprogram we can block the execution of the syscall, setting the *orig_eax* or *orig_rax* value to *-1* as the syscall has not been executed yet at the entry of the syscall, finally, once the syscall finish, we just set in *eax* or *rax* the value -*EPERM* so the operation is not permitted:

```
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);
    }
}
```

In the previous code, we would check for a given syscall if it appears in a blacklist. In the moment before returning we can return another value if we want to emulate for example some behavior.

### Managing Exceptions

With ptrace we can also manage exceptions as we did with the breakpoints and the syscalls, to do that we will have to check for a stop reason equals to **SIGSEGV**, inside of that code, we can get a structure of the exception with **PTRACE_GETSIGINFO**, in here we can get the address where the exception happened, reason, and so on. Let's going to see an example of code:

```
if (WIFSTOPPED(status) && WSTOPSIG(status) == SIGSEGV)
{
    siginfo_t siginfo;

    if (ptrace(PTRACE_GETSIGINFO, pid, NULL, &siginfo) < 0)
    {
        perror("PTRACE_GETSIGINFO");
        exit(-1);
    }

    // get registers from program
    if (ptrace(PTRACE_GETREGS, pid, NULL, &h.pt_reg) < 0)
    {
        perror("PTRACE_GETREGS");
        exit(-1);
    }


    printf("siginfo.si_addr: 0x%08lx\n", siginfo.si_addr);
    printf("Exception in address: 0x%lx\n", h.pt_reg.rip);

    printf("%%rcx: %016x\t%%rdx: %016x\t%%rbx: %016x\n"
                "%%rax: %016x\t%%rdi: %016x\t%%rsi: %016x\n"
                "%%r8:  %016x\t%%r9:  %016x\t%%r10: %016x\n"
                "%%r11: %016x\t%%r12: %016x\t%%r13: %016x\n"
                "%%r14: %016x\t%%r15: %016x\t%%rsp: %016x\n"
                "%%rbp: %016x\n",
                h.pt_reg.rcx, h.pt_reg.rdx, h.pt_reg.rbx,
                h.pt_reg.rax, h.pt_reg.rdi, h.pt_reg.rsi,
                h.pt_reg.r8, h.pt_reg.r9, h.pt_reg.r10,
                h.pt_reg.r11, h.pt_reg.r12, h.pt_reg.r13,
                h.pt_reg.r14, h.pt_reg.r15, h.pt_reg.rsp,
                h.pt_reg.rbp);
}
```