Homework Wet 1

Due date: Thursday, 07/04/16, 12:30

Teaching assistant in charge: Gil Kupfer

**Important:** the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You’re not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding hw3, put them in the hw3 folder

Note: Start working on the assignment as soon as possible!!! This assignment involves algorithmic design, code writing, and extensive testing and debugging.

Postponements can be authorized only by Arthur, the TA in charge. In case you need a postponement, contact him directly.
• Introduction

Your mission in this assignment will be to add a new system call to the interface of the kernel, and to update some existing system calls. While doing so you will gain extra knowledge in compiling the kernel.

Also in this exercise, we will use VMware to simulate a virtual machine on which we will compile and run our "modified" Linux. You will submit only changed source files of the Linux kernel.

• General Description

The developers of the security department of the Linux kernel, decided it would be nice if they could prevent unwanted programs from running. Your mission is to help them doing so.

In the following section you will find a detailed description for what your mission is.

• Detailed Description

You need to implement code wrappers and the corresponding system calls. For example: block_program is a code wrapper and sys_block_program is a system call (see the slides for tutorial 2).

**Code wrappers:**

```c
int block_program(const char *name, unsigned int name_len)
```

**Description:** Block unwanted program from running. After calling this function any execve to the blocked program should failed and recorded. It is allowed to block multiple programs. Blocking a blocked program is legal but make sure you unblock it properly (completely unblock it).

There is no limit on the amount of blocked programs at a given time.

You can assume that the name of the file is bounded by 255 characters. For the sake of simplicity, you can assume that the file name is a full path (starts with /), and it is of the exact path (/root/a.out and not /root ./a.out). It is legal to block program that does not exist yet.

**The return values:**

- On failure: -1
- On success: 0 if the program wasn’t blocked
On failure 'errno' should contain one of following values:

- 'EINVAL' (Invalid argument): name is NULL

**int unblock_program (const char *name, unsigned int name_len)**

Description: Unblock the program. After calling this function the function will be completely unblocked. However, old records of forbidden run tries will not be deleted.

Assumptions as in block_program.

The return values:

- On failure: -1
- On success: 0 if the program was blocked
- 1 if the program wasn’t blocked

On failure 'errno' should contain one of following values:

- 'EINVAL' (Invalid argument): name is NULL

**int is_program_blocked(const char *name, unsigned int name_len)**

Description: Check whether a program is blocked.

Assumptions as in block_program.

The return values:

- On failure: -1
- On success: 0 if the program wasn’t blocked
- 1 if the program was blocked

On failure 'errno' should contain one of following values:

- 'EINVAL' (Invalid argument): name is NULL

**int get_blocked_count(void)**

Description: Get how many programs are currently blocked.

The return values:

- On failure: doesn’t fail
- On success: the number of currently blocked programs

**int get_forbidden_tries(int pid, char log[][256], unsigned int n)**
Description: For a given process \((\text{pid})\), get a list of the blocked programs it tried to execve to. Each time we recognize a try to run a blocked program we fail it and log this access to the task which tried to run it. This function retrieves this log ordered by the time of access. Each entry is the filename as it was given to block_program.

If there are more than \(n\) accesses, the log will contain the first \(n\) ones.

The return values:

On failure: -1

On success: The number of accesses actually written (may be less than \(n\))

On failure 'errno' should contain one of following values:

- 'EINVAL' (Invalid argument): \(n\) is 0
- 'ESRCH' (No such process): No such process exists
- 'EFAULT' (Bad address): incorrect data address for data (the log array can’t be written to (e.g. doesn’t belong to the calling process address space).
The System calls:

- **int sys_block_program(const char *name, unsigned int name_len)**
  (system call #243)
  Return values:
  - On success: 0 if the program wasn’t blocked
    1 if the program was blocked
  - ‘-EINVAL’ (Invalid argument): name is NULL or name_len is 0

- **int sys_unblock_program(const char *name, unsigned int name_len)**
  (system call #244)
  Return values:
  - On success: 0 if the program was blocked
    1 if the program wasn’t blocked
  - ‘-EINVAL’ (Invalid argument): name is NULL or name_len is 0

- **int sys_is_program_blocked(const char *name, unsigned int name_len)**
  (system call #245)
  Return values:
  - On success: 1 if the program was blocked
    0 if the program wasn’t blocked
  - ‘-EINVAL’ (Invalid argument): name is NULL or name_len is 0

- **int sys_get_blocked_count(void)**
  (system call #246)
  Return values:
  - On success: the number of currently blocked programs

- **int sys_get_forbidden_tries (int pid, char log[][256], unsigned int n)**
  (system call #247)
  Return values:
  - On success: The number of accesses actually written
  - ‘-EINVAL’ (Invalid argument): n is 0
- `ESRCH` (No such process): No such process exists
- `EFAULT` (Bad address): Incorrect data address for data (the log array can’t be written to (e.g. doesn’t belong to the calling process address space).
Code Wrapper

As was mentioned earlier, you need to implement a code wrapper for your system calls.

Below is an example of the code wrapper for `my_system_call (#244)`. Follow this example to write the wrappers.

```c
int my_system_call (int p1, char *p2, int p3)
{
    unsigned int res;
    __asm__
    {
        "int $0x80;"
        : "=a" (res)
        : "0" (244), "b" (p1), "c" (p2), "d" (p3)
        : "memory";
    }
    if (res >= (unsigned long)(-125))
    {
        errno = -res;
        res = -1;
    }
    return (int) res;
}
```

Explanation of inline assembler:
The assembler structure is:

```
asm ( assembler template
    : output operands (optional)
    : input operands (optional)
    : clobber list (optional)
);
```

The asm is volatile to tell the compiler it has side effect besides the output operands, In our case it means that even if res is never used the compiler may not delete this assembly block.

The only command we need to issue in assembly is "int $0x80"
The rest of the preparation is done by the compiler assuming we describe the operand correctly.

The operands are numbered according to the order that they specified and are described below

**Output operands:**
%0: "=a" (res)-the "=" means it output and "a" means it should be in the register eax, the (res) say that it should be put in the variable res.

**Input operands:**
%1: "0" (244) – the "0" say we want this operand to have the same constraints as operand %0, which in our case mean to be in eax.
The (244) says we want it to have the value 244 when the assembly block begins.
%2: "b" (p1) – "b" means we want this operand to be in ebx and (p1) means we want it to have the value of p1
%3: "c" (p2) – "c" means we want this operand to be in ecx and (p2) means we want it to have the value of p2
%4: "d" (p3) – "d" means we want this operand to be in edx and (p3) means we want it to have the value of p3

clobber list:
"memory" tells the compile that the asm block may write to memory that wasn’t specified as an output operand.

Useful links:
http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

• Notes and Tips

• Save the access log for a given process in the process descriptor struct task_struct, Its definition can be found in sched.h
• You are not allowed to use syscall functions to implement code wrappers, or to write the code wrappers for your system calls using the macro _syscall1. You should write the code wrappers according to the example of the code wrapper given above.
• You should use the functions copy_from_user, copy_to_user, etc. in order to copy data from user space to kernel space and vice versa.
  Useful link: User space memory access from the Linux kernel (skip to Kernel APIs section)
• All your changes must be made in the kernel level.
• Submit only modified files from the Linux kernel.
• You should not print the code.
• Start working on the assignment as soon as possible. The deadline is final, NO postponements will be given, and high load on the VMWare machines will not be accepted as an excuse for late submissions.
• ’-ENOENT’ stands for ‘minus ENOENT’
• If there are more than one error when calling a syscall you should return the first one by the order in the function description.
• Don’t forget to set initialize all your data structures.
• Write your own tests. We will check your assignment also with our test program.
We are going to check for kernel oops (errors that don’t prevent the kernel from continue running such as NULL dereference in syscall implementation). You should not have any. If there was kernel oops, you can see it in dmesg (dmesg it’s the command that prints the kernel messages, e.g. printk, to the screen).

To read it in more conveniently: `dmesg | less -S`

- Linux is case-sensitive. `entry.S` means `entry.S`, not `Entry.s`, `Entry.S` or `entry.s`.
- You can assume that the system is with a single CPU.
- You should release all the allocated memory for the process when it ends. Look at the function `do_exit`.
- You should use `kmalloc` and `kfree` in the kernel in order to allocate and release memory. If `kmalloc` fails you should return `ENOMEM`. For the `kmalloc` function use flag `GFP_KERNEL` for the memory for kernel use.
- Pay attention that the process descriptor size is limited. Do not add to many new fields. Also, add your fields at the end of the struct because the kernel sometimes uses the offsets of the fields.
- You may want to look on `/usr/src/linux-2.4.18-14custom/include/linux/list.h` and tutorial 3 for already implemented list.
- If you need global variables, you can put them in `kernel/blocker.c` (The file with your syscalls)

**What should you do?**

Use VMware, like you learned in the preliminary assignment, in order to make the following changes in the Linux kernel:

1. Put the implementation of the new system calls in the file `kernel/blocker.c` that you will have to create and add to the kernel. Update the `makefile` in that directory to compile your new file too. (Tip: add it to `obj-y`).
2. Update `sched.h` (constant + new fields definition)
3. Update `entry.S` (add system call number)
4. Make any necessary changes in the kernel code so the new system calls can be used like any other existing Linux system call. Your changes can include modifying any `.c`, `.h` or `.S` (assembly) file that you find necessary.
5. Find the function in `sched.c` that is responsible for initializing first process task structure and initialize the log there.
6. Make necessary changes in file `fork.c` and `exit.c`
7. Updating more files is needed.
8. Recompile and run the new kernel like you did in the preliminary assignment.
9. Put the wrappers functions in `blocker.h`
10. Boot with your new Linux, and try to compile and run the test program to make sure the new system calls work as expected.

Did it all? Good work, Submit your assignment. 😊

**Submission**

The submission of this assignment has two parts:

- An electronic submission – you should create a zip file *(use zip only, not gzip, tar, rar, 7z or anything else)* containing the following files:
  a. A tarball named `kernel.tar.gz` containing all the files in the kernel that you created or modified (including any source, assembly or makefile) + your code wrappers.

To create the tarball run (inside VMWare):

```bash
cd /usr/src/linux-2.4.18-14custom
tar -czf kernel.tar.gz <list of modified or added files>
```

For example, if the only files you changed are `arch/i386/kernel/entry.S` and `kernel/blocker.c` you should run (the second command should be on one line, it was split due to being too long on paper):

```bash
cd /usr/src/linux-2.4.18-14custom
tar --czf kernel.tar.gz arch/i386/kernel/entry.S kernel/blocker.c
```

Make sure you don't forget any file and that you use relative paths in the tar command, i.e., use `kernel/blocker.c` and not `/usr/src/linux-2.4.18-14custom/kernel/blocker.c`

**Important:** If you missed a file and because of this the exercise is not working you will get 0 and resubmission will cost 10 points. In case you missed an important file (such as the file with all you logic) we may not accept it at all.

In order to prevent it you should open the tar on your host machine and see that the files are structured as they supposed to be in the source directory.
It is highly recommended to create another clean copy of the guest machine and open the tar there and see it behave as you expected.

To open the tar:

```
cd /usr/src/linux-2.4.18-14custom
tar -xzf <path to tarball>/kernel.tar.gz
```

b. A file named `blocker.h` that contains the implementation of your wrapper functions and other declarations.

c. A file named `submitters.txt` which includes the ID, name and email of the participating students. The following format should be used:

```
Bill Gates bill@t2.technion.ac.il 123456789
Linus Torvalds linus@gmail.com 234567890
```

**Important Note:** Make the outlined zip structure **exactly**. In particular, the zip should contain only the following files (no directories!): `kernel.tar.gz` and `submitters.txt`. You can create the zip by running (inside VMware):

```
zip final.zip kernel.tar.gz blocker.h submitters.txt
```

The zip should look as follows:

```
zipfile -->
 |    --> kernel.tar.gz
 |    --> submitters.txt
 |    --> blocker.h
```

Good Luck,
The course staff