Changing memory protection in an arbitrary process.

perception-point.io/changing-memory-protection-in-an-arbitrary-process

November 28, 2018

Recently, we faced this very specific task: changing the protection flags of memory regions in an arbitrary process. As this task may seem trivial, we encountered some obstacles and learned new things in the process, mostly about Linux mechanisms, memory protection and kernel development. Here is a brief overview of our work, including three approaches we took and what made us seek for a better solution each time.

Introduction to mprotect.

In modern operating systems, each process has its own virtual address space (a mapping from virtual addresses to physical addresses). This virtual address space consists of memory pages (contiguous memory chunks of some fixed size), and each page has protection flags which determine the kind of access allowed to this page (Read, Write & Execute). This mechanism relies on the architecture page tables (fun fact: in the x64 architecture, you can't make a page write-only, even if you specifically request it from your operating system – it will always be readable as well).

In Windows, you can change the protection of a memory region with the API functions **VirtualProtect** or **VirtualProtectEx.** The latter makes our task very easy: its first argument, hProcess, is "a handle to the process whose memory protection is to be changed" (from MSDN).

Linux Memory Protection

In Linux, on the other hand, we're not so lucky: the API to change memory protection is the system calls **mprotect** or **pkey_mprotect**, and both always operate on the current process' address space. We'll review now our approaches to solve this task in Linux on x64 architecture (we assume root privileges).

APPROACH ONE

mprotect Code Injection.

Well, if mprotect always acts on the current process, we need to make our target process call it from its own context. This is called code injection, and it's achievable in many different ways. We chose to implement it with the **ptrace** mechanism, which lets one process "observe and control the execution of another process" (from the man page), including the ability to change the target process' memory and registers. This mechanism is used in debuggers (like gdb) and tracing utilities (like strace). An outline of the steps required to inject code using ptrace:

1. Attach to the target process with ptrace. If there are multiple threads in the process, it may be wise to stop all the other threads as well.
2. Find an executable memory region (by examining /proc/PID/maps) and write there the opcode *syscall* (hex: 0f 05).
3. Modify the registers according to the calling convention: first, change rax to the system call number of mprotect (which is 10). Then, the first three arguments (which are the start address, the length and the protection desired) are stored in rdi, rsi, and rdx respectively. Finally, change rip to the address used in step 2.
4. Resume the process until the system call returns (ptrace allows you to trace enters and exits of system calls).
5. Recover the overridden memory and registers, detach from the process and resume its normal execution.

This approach was our first and most intuitive one, and worked great until we discovered another mechanism in Linux which completely ruined it: **seccomp**. Basically, it's a security facility in the Linux kernel which allows a process to enter itself into some kind of a "jail", where it can't call any system call besides read, write, _exit and sigreturn. There is also an option to specify arbitrary system calls and their arguments to filter only them.

Therefore, if a process enabled seccomp mode and we try inject a call to mprotect into it, then the kernel will kill the process as it is not allowed to use this system call. We wanted to be able to act on these processes as well, so the search for a better solution continues…

APPROACH TWO

Imitate mprotect in a kernel module.

The seccomp problem eliminated every solution from the process' user mode, hence the next approach certainly resides in kernel mode. In the Linux kernel, each thread (both user threads and kernel threads) is represented by a structure named task_struct, and the current thread (task) is accessible through the pointer *current*. The internal implementation of mprotect in the kernel uses the pointer *current*, so our first thought was – let's just copy-paste the code of mprotect to our kernel module, and replace each occurrence of *current* with a pointer to our target thread's task_struct. Right?

Well, as you may have guessed, copying C code is not so trivial – there's a heavy use of unexported functions, variables and macros which we just cannot access. Some functions declarations are exported in the header files, but their actual addresses aren't exported by the kernel. This specific problem can be solved if the kernel was compiled with kallsyms support, and then it exports all of its internal symbols through the file /proc/kallsysm.

Despite these problems, we tried to implement only the essence of mprotect, even solely for educational purposes. So we headed to write a kernel module which gets the target PID and the parameters to mprotect, and imitates its behaviour. First, we need to obtain the desired memory mapping object, which represents the address space of the thread:

```
    /* Find the task by the pid */
    pid_struct = find_get_pid(params.pid);
    if (!pid_struct)
        return -ESRCH;

    task = get_pid_task(pid_struct, PIDTYPE_PID);
    if (!task) {
        ret = -ESRCH;
        goto out;
    }

    /* Get the mm of the task */
    mm = get_task_mm(task);
    if (!mm) {
        ret = -ESRCH;
        goto out;
    }

    …
    …

out:
    if (mm) mmput(mm);
    if (task) put_task_struct(task);
    if (pid_struct) put_pid(pid_struct);
```

Now that we have the memory mapping object, we need to dig deeper. The Linux kernel implements an abstraction layer to manage memory regions, each region is represented by the structure vm_area_struct. To find the correct memory region, we use the function **find_vma** which searches the memory mapping by the desired address.

The vm_area_struct contains the field vm_flags which represents the protection flags of the memory region in an architecture-independent manner, and vm_page_prot which represents it in an architecture-dependent manner. Changing these fields alone won't really affect the page table (but will affect the output of /proc/PID/maps, we tried it!). You can read more about it here.

After some reading and digging into the kernel code, we detected the most essential work needed to really change the protection of a memory region:

1. Change the field vm_flags to the desired protection.
2. Call the function **vma_set_page_prot_func** to update the field vm_page_prot according to the vm_flags field.

3. Call the function **change_protection_func** to actually update the protection bits in the page table.

This code works, but it has many problems – first, we implement only the essential parts of mprotect, but the original function does much more than we did (for example, splitting and joining memory regions by their protection flags). Second, we use two internal functions which are not exported by the kernel (vma_set_page_prot_func and change_protection_func). We can call them using kallsyms, but this is prone to troubles (perhaps their names will be changed in the future, or maybe the whole internal implementation of memory regions will be altered). We wanted a more generic solution which doesn't take internal structures into consideration, so the search for a better solution continues…

APPROACH THREE

Using the target process's memory mapping.

This approach is very similar to the first one – there, we wanted to execute code in the context of the target process. Here, instead, we execute code in our own thread, but we use the "memory context" of the target process, meaning: we use its address space.

Changing your address space is possible in kernel mode through several API functions, of them we will use **use_mm**. As the documentation clearly specifies, "this routine is intended to be called only from a kernel thread context". These are threads which are created in the kernel and do not need any user address space, so it's fine to change their address space (the kernel's region inside the address space is mapped the same way in every task).

One easy way to run your code in a kernel thread is the work queue interface of the kernel, which allows you to schedule a work with a specific routine and specific arguments. Our work routine is very minimal – it gets the memory mapping object of the desired process and the parameters to mprotect, and does the following (do_mprotect_pkey is the internal function in the kernel that implements the mprotect and pkey_mprotect system calls):

```
use_mm(suprotect_work->mm);
suprotect_work->ret_value = do_mprotect_pkey(suprotect_work->start,
                                             suprotect_work->len,
                                             suprotect_work->prot, -1);
unuse_mm(suprotect_work->mm);
```

When our kernel module gets a request to change protection in some process (through a special IOCTL), it first finds the desired memory mapping object (as we explained in the previous approach) and then just schedules the work with the right parameters.

This solution still has one minor problem – the function do_mprotect_pkey_func isn't exported by the kernel and needs to be fetched using kallsyms. Unlike the former solution, this internal function is not very prone to changes as it's tied to the system call pkey_mprotect, and we don't handle internal structures, hence we can call it only a "minor problem".

We hope you found some interesting information and techniques in this post. If you're interested, the source code of this proof-of-concept kernel module is available in our github here.