Talos Vulnerability Report

TALOS-2020-1211

Linux Kernel /proc/pid/syscall information disclosure vulnerability

APRIL 27, 2020

CVE NUMBER

CVE-2020-28588

Summary

An information disclosure vulnerability exists in the /proc/pid/syscall functionality of Linux Kernel 5.1 Stable and 5.4.66. More specifically, this issue has been introduced in v5.1-rc4 (commit 631b7abacd02b88f4b0795c08b54ad4fc3e7c7c0) and is still present in v5.10-rc4, so it's likely that all versions in between are affected. An attacker can read /proc/pid/syscall to trigger this vulnerability, which leads to the kernel leaking memory contents.

Tested Versions

Linux Kernel v5.10-rc4
Linux Kernel v5.4.66
Linux Kernel v5.9.8

Product URLs

https://github.com/torvalds/linux

CVSSv3 Score

4.0 - CVSS:3.0/AV:L/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N

CWE

CWE-681 - Incorrect Conversion between Numeric Types

Details

The Linux Kernel is the free and open-source core of Unix-like operating systems.

The Linux kernel provides a pseudo-filesystem called `proc` which allows for interfacing with various kernel data structures. `procfs` is usually mounted at `/proc` and exposes several entries in the form of files and directories, which can be read and/or written to, allowing to read and write kernel variables.

For the present vulnerability, we speak in particular of the simple `/proc/<pid>/syscall` `procfs` entry. This file only supports being read, and we can see the output on any given Linux system who's kernel was configured with `CONFIG_HAVE_ARCH_TRACEHOOK`:

```
[<_<]:> uname -a
Linux ubuntu 5.4.0-53-generic #59-Ubuntu SMP Wed Oct 21 09:38:44 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

[>_>]:> cat /proc/self/syscall
0 0x3 0x7f7766aad000 0x20000 0x22 0x7f7766aac010 0x0 0x7ffd6031d8f8 0x7f776714f142
```

While this output might be known or guessable to some, for completeness we now look at `man proc` for an overview:

```
/proc/[pid]/syscall (since Linux 2.6.27)
    This file  exposes the system call number and argument registers for the
    system call currently being executed by the process, followed by the
    values of the stack pointer and program counter registers.  The values
    of all six argument registers are exposed, although most system calls
    use fewer registers.

    If the process is blocked, but not in a system call, then the file
    displays -1 in place of the system call number, followed by just the
    values of the stack pointer and program counter.  If process is not
    blocked, then the file contains just the string "running".

    This file is present only if the kernel was configured with
    CONFIG_HAVE_ARCH_TRACEHOOK.

    Permission to access this file is governed by a ptrace access mode
    PTRACE_MODE_ATTACH_FSCREDS check; see ptrace(2).
```

So, to reiterate, `/proc/<pid>/syscall` outputs the register state of a given process if it is blocking at the time of `/proc/<pid>/syscall` being read. For further detail we examine the implementation inside `<linux_kernel>/fs/proc/base.c` in the `proc_pid_syscall` function:

```
static int proc_pid_syscall(struct seq_file *m, struct pid_namespace *ns,
                struct pid *pid, struct task_struct *task)
{
    struct syscall_info info;
    u64 *args = &info.data.args[0];
    int res;

    res = lock_trace(task);
    if (res)
        return res;

    if (task_current_syscall(task, &info))                    // [1]
        seq_puts(m, "running\n");
    else if (info.data.nr < 0)
        seq_printf(m, "%d 0x%llx 0x%llx\n",
                info.data.nr, info.sp, info.data.instruction_pointer);
    else
        seq_printf(m,                                         // [2]
                "%d 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx\n",
                info.data.nr,
                args[0], args[1], args[2], args[3], args[4], args[5],
                info.sp, info.data.instruction_pointer);
    unlock_trace(task);

    return 0;
}
```

At [1], the register information of the current process is gathered into the `syscall_info info` structure, and then at [2] we proceed to output the register state. For completeness, the `syscall_info` structure looks like so:

```
struct syscall_info {
    __u64           sp;
    struct seccomp_data data;
};

struct seccomp_data {
    int nr;
    __u32 arch;
    __u64 instruction_pointer;
    __u64 args[6];
};
```

It's mainly just important to note that the `data.args` array consists of `__u64` sized slots. Continuing on, we look at how this structure gets populated with data within `task_current_syscall`:

```
int task_current_syscall(struct task_struct *target, struct syscall_info *info) {
    long state;
    unsigned long ncsw;

    if (target == current)
        return collect_syscall(target, info);        // [1]

    state = target->state;
    if (unlikely(!state))
        return -EAGAIN;

    ncsw = wait_task_inactive(target, state);
    if (unlikely(!ncsw) ||
        unlikely(collect_syscall(target, info)) ||   // [2]
        unlikely(wait_task_inactive(target, state) != ncsw))
        return -EAGAIN;

    return 0;
}
```

From this function we then hit `collect_syscall` at either [1] or [2]:

```
static int collect_syscall(struct task_struct *target, struct syscall_info *info)
{
    struct pt_regs *regs;

    if (!try_get_task_stack(target)) {
        /* Task has no stack, so the task isn't in a syscall. */
        memset(info, 0, sizeof(*info));
        info->data.nr = -1;
        return 0;
    }

    regs = task_pt_regs(target);
    if (unlikely(!regs)) {
        put_task_stack(target);
        return -EAGAIN;
    }

    info->sp = user_stack_pointer(regs);                        // [1]
    info->data.instruction_pointer = instruction_pointer(regs);   // [2]

    info->data.nr = syscall_get_nr(target, regs);               // [3]
    if (info->data.nr != -1L)
        syscall_get_arguments(target, regs,                     // [4]
                      (unsigned long *)&info->data.args[0]);

    put_task_stack(target);
    return 0;
}
```

At [1], the `$sp` register is populated, at [2], `$pc` is populated, and also at [3] the syscall number is read in. All that's left to gather before printing results is the first five general registers, which is done at [4] inside the arch-specific (ARM in this instance) `syscall_get_arguments` function:

```
// arch/arm/include/asm/syscall.h
static inline void syscall_get_arguments(struct task_struct *task,
                                         struct pt_regs *regs,
                                         unsigned long *args)     // [1]
{
    args[0] = regs->ARM_ORIG_r0;
    args++;

    memcpy(args, &regs->ARM_r0 + 1, 5 * sizeof(args[0])); // [2]
}
```

In looking at this specific function, everything looks fine, but it's worth noting that the `args` parameter passed in came all the way from the `proc_pid_syscall` function, and as such is actually of type `__u64 args[6]`. On an ARM system, the function definition at [1] casts the size of the `arg` array to four bytes elements from eight bytes (since `unsigned long` in ARM is 4 bytes) resulting in the `memcpy` at [2] copying in 20 bytes (plus 4 for `args[0]`).

Similarly for i386, where `unsigned long` is 4 bytes, only the first 24 bytes of the `args` argument are written to, leaving the remaining 24 bytes untouched:

```
// arch/x86/include/asm/syscall.h
#ifdef CONFIG_X86_32

static inline void syscall_get_arguments(struct task_struct *task,
                                         struct pt_regs *regs,
                                         unsigned long *args)
{
    memcpy(args, &regs->bx, 6 * sizeof(args[0]));
}
```

In both cases, if we look back at the `proc_pid_syscall` function however, we can see the following format string is used for output:

```
seq_printf(m,
    "%d 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx 0x%llx\n",
    info.data.nr,
    args[0], args[1], args[2], args[3], args[4], args[5],
    info.sp, info.data.instruction_pointer);
```

While on 32-bit ARM and i386 we only copy in 24 bytes into the `args` array, the format string ends up reading 48 bytes from the `args` array since the `%llx` format string is eight bytes on both 32-bit and 64-bit systems. Thus, 24 bytes of uninitialized stack memory end up getting output, which could lead to a KASLR bypass.

We first discovered this issue on the Azure Sphere device (version 20.10), a 32-bit ARM device that runs a patched Linux kernel:

```
> uname -a
Linux (none) 5.4.66-mt3620-azure-sphere #1 Fri Oct 2 20:28:48 UTC 2020 armv7l GNU/Linux
> cat /proc/self/syscall
0 0x300000001 0x100000000000000 0x1000000 0xc0201900c0009d84 0xbf8752f100000004 0x1000c0201900 0xbee4bd28 0x962c8d0c
```

Indeed, we can see addresses (0xc0201900, 0xc0009d84) that reference the kernel space. If we proceed to cat out this entry during reboot:

```
> cat /proc/self/syscall
0 0x300000001 0x100000000000000 0x1000000 0xc0201900 0xbf8752e7bf88c041 0x1000c0201900 0xbeef0d28 0x888cbd0c
> cat /proc/self/syscall
0 0x300000001 0x100000000000000 0x1000000 0xc0009d84 0x400cc0c0127900 0xc0169db0a0000013 0xbef3bd28 0x8f1c9d0c
> cat /proc/self/syscall
0 0x300000001 0x100000000000000 0x1000000 0xc0201900c0009d84 0xbf8752f100000004 0x1000c0201900 0xbebd3d28 0x457c7d0c
```

We can see above that the memory does indeed change (depending on what all else is going on in the kernel).
Lastly, if we cat out two different processes' `/proc/pid/syscall` (during a period of inactivity) we can see the same 24 bytes of data leaked from the kernel stack:

```
> cat /proc/22/syscall
0 0xbedeb04000000003 0xbedeb03c 0x0 0xc0201900c0009d84 0xbf8752f100000004 0x1000c0201900 0xbede9d70 0x34cc93a6
> cat /proc/self/syscall
0 0x300000001 0x100000000000000 0x1000000 0xc0201900c0009d84 0xbf8752f100000004 0x1000c0201900 0xbeeaed28 0x52acbd0c
```

In general, to trigger this memory leak more frequently, it's enough to execute these commands in different shells:

```
# echo 0 > /proc/sys/kernel/randomize_va_space        # only needed for a cleaner output
$ while true; do cat /proc/self/syscall; done | uniq  # waits for changes
$ while true; do free &>/dev/null; done               # triggers changes
```

For root cause, we posit that commit 631b7abacd02b88f4b0795c08b54ad4fc3e7c7c0 introduced this issue, meaning that this issue has been present since v5.1-rc4.

Timeline

2020-11-25 - Vendor Disclosure
2020-12-03 - Patch merged
2020-02-17 - Talos follow-up
2020-02-17 - Maintainer confirms that patch merged on 2020-12-03
2021-04-27 - Public Disclosure

CREDIT

Discovered by Lilith >_> and Claudio Bozzato of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2020-1052

NEXT REPORT TALOS-2020-1214