```
Date: Tue, 24 Mar 2020 22:50:49 +0100
From: Adam Zabrocki <pi3@....com.pl>
To: linux-kernel@...r.kernel.org, kernel-hardening@...ts.openwall.com
Subject: Curiosity around 'exec_id' and some problems associated with it

Problems:
     I.               The "exit_notify" function in the Linux kernel does not
                      sufficiently restrict exit signals
    II.               Preserve references to the "old" / dead process' VM via
                      file descriptor
Author:               Adam Zabrocki (<pi3@....com.pl>)
Date:                 September 1999 - March 2020


   Description:

In 2009 Oleg Nesterov discovered that Linux kernel has an incorrect logic to
reset ->exit_signal. As a result, the malicious user can bypass it if it execs
the setuid application before exiting (->exit_signal won't be reset to SIGCHLD).
His original message can be found here:

https://marc.info/?l=linux-kernel&m=123560588713763&w=2

CVE-2009-1337 was assigned to track this issue. More information about this bug
can be found here:

https://www.cvedetails.com/cve/CVE-2009-1337/

Patch:

https://lore.kernel.org/patchwork/patch/150993/

The logic responsible for handling ->exit_signal has been changed a few times
and the current logic is locked down since Linux kernel 3.3.5. However, it is
not fully robust and it's still possible for the malicious user to bypass it.
Basically, it's possible to send arbitrary signals to a privileged (suidroot)
parent process (Problem I.). Nevertheless, it's not trivial and more limited
comparing to the CVE-2009-1337.


   Details (Problem I.):

    When process dies function do_exit() -> exit_notify() from "kernel/exit.c"
is called:

"kernel/exit.c"
static void exit_notify(struct task_struct *tsk, int group_dead)
{
        ...
        tsk->exit_state = EXIT_ZOMBIE;
        if (unlikely(tsk->ptrace)) {
        ...
        } else if (thread_group_leader(tsk)) {
                autoreap = thread_group_empty(tsk) &&
                        do_notify_parent(tsk, tsk->exit_signal);
        } else {
        ...
    }
    ...
}

To be able to inform that child process died, do_notify_parent() from
"kernel/signal.c" is executed:

"kernel/signal.c"
bool do_notify_parent(struct task_struct *tsk, int sig)
{
        ...
        if (sig != SIGCHLD) {
                /*
                 * This is only possible if parent == real_parent.
                 * Check if it has changed security domain.
                 */
                if (tsk->parent_exec_id != tsk->parent->self_exec_id)
                        sig = SIGCHLD;
        }
        ...
        if (valid_signal(sig) && sig)
                __group_send_sig_info(sig, &info, tsk->parent);
        ...
}

It is possible for the child process to send to the parent process any signal
only when they run within the same security domain. If parent or child process
are not in the same domain, kernel will overwrite signal with SIGCHLD value.

However, this check is weak. It is possible for the malicious user to cause an
integer overflow for the value tsk->parent->self_exec_id and bypass this
validation.

Both values, self_exec_id and parent_exec_id are defined in the "task_struct"
structure in file "include/linux/sched.h":

"include/linux/sched.h"
struct task_struct {
        ...
        /* Thread group tracking: */
        u32                             parent_exec_id;
        u32                             self_exec_id;
        ...
}

Linux kernel defines "u32" as "unsigned int" type, which most of the modern
compiler's data model defines as a 32 bits value.

Some curiosities which are interesting to point out:

 1) Linus Torvalds in 2012 suspected that such 'overflow' might be possible.
    You can read more about it here:

    https://www.openwall.com/lists/kernel-hardening/2012/03/11/4

 2) Solar Designer in 1999(!) was aware about the problem that 'exit_signal' can
    be abused. The kernel didn't protect it at all at that time. So he came up
    with the idea to introduce those two counters to deal with that problem.
    Originally, these counters were defined as "long long" type. However, during
    the revising between September 14 and September 16, 1999 he switched from
    "long long" to "int" and introduced integer wraparound handling. His patches
    were merged to the kernel 2.0.39 and 2.0.40.

 3) It is worth to read the Solar Designer's message during the discussion about
    the fix for the problem CVE-2012-0056 (I'm referencing this problem later in
    that write-up about "Problem II"):

    https://www.openwall.com/lists/kernel-hardening/2012/03/11/12


   Exploitability:
```

    To be able to cause an integer overflow on self_exec_id or parent_exec_id, attacker needs to find a way to precisely control that value from the user-mode. Linux kernel references '*_exec_id' variables just in a few places in the code:

    1) Function do_notify_parent() from "kernel/exit.c"
    2) Function copy_process() from "kernel/fork.c"
    3) Function setup_new_exec() from "fs/exec.c"

Option 1) we already covered. Let's take a look at 2nd case:

```
"kernel/fork.c"
static __latent_entropy struct task_struct *copy_process(
                                        struct pid *pid,
                                        int trace,
                                        int node,
                                        struct kernel_clone_args *args)
{
        ...
        /* CLONE_PARENT re-uses the old parent */
        if (clone_flags & (CLONE_PARENT|CLONE_THREAD)) {
                p->real_parent = current->real_parent;
                p->parent_exec_id = current->parent_exec_id;
        } else {
                p->real_parent = current;
                p->parent_exec_id = current->self_exec_id;
        }
        ...
}
```

When mother creates a child process / thread (or in general 'task'), there is a way to semi-control 'parent_exec_id' value by controlling which process will be assigned as a 'real_parent'. To be able to do that, an attacker can pass (or not) a CLONE_PARENT or CLONE_THREAD flag. However, this 'primitive' doesn't directly allow to bypass the checks in do_notify_parent() function.

The only hope is in option 3:

```
"fs/exec.c"
void setup_new_exec(struct linux_binprm * bprm)
{
        ...
        current->self_exec_id++;
        flush_signal_handlers(current, 0);
}
```

Whenever current process executes a new binary, kernel increments "self_exec_id" variable. There is no validation on the current value, and malicious user can indirectly set any value for that variable.

However, to be able to bypass validation in do_notify_parent() function, attacker needs to control parent's "self_exec_id":

```
                if (tsk->parent_exec_id != tsk->parent->self_exec_id)
                        sig = SIGCHLD;
```

To achieve that, malicious user must:

 a) Create a mother process
 b) Invoke clone() without CLONE_PARENT neither CLONE_THREAD flag. In such case, a new process will get the same values for "parent_exec_id" and "self_exec_id" as mother process.
 c) Mother process executes itself as many times as to be able to overflow own "self_exec_id" value, and set it to the "original self_exec_id" - 1.
 d) If desired value is acquired, then mother process executes any SUID privileged binary. By doing that, mother process will run in much higher security domain but the child's "parent_exec_id" will match "tsk->parent->self_exec_id" and that leads to bypass of the validation in do_notify_parent() function.

It is important to note, that beginning from the version 3.3.5, before kernel calls setup_new_exec(), function flush_old_exec -> de_thread() will be executed:

```
"fs/exec.c"
static int de_thread(struct task_struct *tsk)
{
        ...
no_thread_group:
        /* we have changed execution domain */
        tsk->exit_signal = SIGCHLD;
        ...
}
```

This function overwrites "exit_signal" to always be SIGCHLD. This means that any "exit_signal" set-up by the mother to the new child process, will be overwriten as soon as child execs.

A few examples:

1) No execve at all

```
     tsk1: mother
,--------------------,
|  self_exec_id = 6  |
| parent_exec_id = 6 |
|   parent = <bash>  |
|   exit_signal =    |
|           SIGCHLD  |
`--------------------`
  ^       |
  |       |
  |       `---> clone(SIGSEGV) --->     tsk2: child
  |                                  ,--------------------,
  |                                  |  self_exec_id = 6  |
  |                                  | parent_exec_id = 6 |
  |                                  |    parent = tsk1   |
  |                                  |   exit_signal =    |
  |                                  |           SIGSEGV  |
  |   original                       `--------------------`
  | "exit_signal"                              |
  |    SIGSEGV                                 v
  |                                        exit(...)
  |                                            |
  |                                            v
  |                                    x = tsk2->parent_exec_id = 6
  |                                    y = tsk2->tsk1->self_exec_id = 6
  |                                         x == y
  |                                            |
  |                                            v
  ^------------------------< Deliver original "exit_signal" (SIGSEGV) >
```

2) child calls execve

```
     tsk1: mother
,--------------------,
|  self_exec_id = 6  |
| parent_exec_id = 6 |
|   parent = <bash>  |
|   exit_signal =    |
|           SIGCHLD  |
`--------------------`
  ^       |
  |       |
  |       `---> clone(SIGSEGV) --->     tsk2: child
  |                                  ,--------------------,
  |                                  |  self_exec_id = 6  |
  |                                  | parent_exec_id = 6 |
  |                                  |    parent = tsk1   |
  |                                  |   exit_signal =    |
  |                                  |           SIGSEGV  |
```

```
  |   original                      `--------------------`
  | "exit_signal"                              |
  |    SIGCHLD                                 v
  |                                        execve(...)
  |                                    exit_signal = SIGCHLD
  |                                            |
  |                                            v
  |                                        exit(...)
  |                                            |
  |                                            v
  |                                    x = tsk2->parent_exec_id = 6
  |                                    y = tsk2->tsk1->self_exec_id = 6
  |                                         x == y
  |                                            |
  |                                            v
  ^--------------------------< Deliver overwritten "exit_signal" (SIGCHLD) >
```

3) mother calls execve

```
     tsk1: mother
,--------------------,
|  self_exec_id = 6  |
| parent_exec_id = 6 |
|   parent = <bash>  |----------------------,
|   exit_signal =    |                      |
|           SIGTERM  |                      |
`--------------------`                      |
          |                                 |
          v                                 |
       execve(...)                          |
          |                                 v
          v                            clone(SIGSEGV)
     tsk1: mother                           |
,--------------------,                      |
|  self_exec_id = 7  |                      |
| parent_exec_id = 6 |                      |
|   parent = <bash>  |                      |
|   exit_signal =    |                      |
|           SIGCHLD  |                      |
`--------------------`                      |
                                            |
                                            |
                                            |
                                            |
                                            v
                                        tsk2: child
                                    ,--------------------,
                                    |  self_exec_id = 6  |
                                    | parent_exec_id = 6 |
                                    |    parent = tsk1   |
                                    |   exit_signal =    |
                                    |           SIGSEGV  |
                                    `--------------------`
                                               |
                                               v
                                           execve(...)
                                       exit_signal = SIGCHLD
                                               |
                                               v
                                           exit(...)
                                               |
                                               v
                                       x = tsk2->parent_exec_id = 6
                                       y = tsk2->tsk1->self_exec_id = 7
                                            x != y
                                               |
                                               v
                                "exit_signal" will NOT be delivered
```

4) mother execs as many times as generates an integer overflow (our attack)

```
     tsk1: mother
,--------------------,
|  self_exec_id = 6  |
| parent_exec_id = 6 |
|   parent = <bash>  |----------------------,
|   exit_signal =    |                      |
|           SIGTERM  |                      |
`--------------------`                      |
          |                                 |
          v                                 |
       execve(...)                          |
          |                                 v
          v                            clone(SIGSEGV)
     tsk1: mother                           |
,--------------------,                      |
|  self_exec_id = 7  |                      |
| parent_exec_id = 6 |                      |
|   parent = <bash>  |                      |
|   exit_signal =    |                      |
|           SIGCHLD  |                      |
`--------------------`                      |
          |                                 |
          v                                 |
       execve(...)                          |
          |                                 |
         ...                                |
    "self_exec_id"                          |
      overflows                             |
         ...                                |
          |                                 |
          v                                 |
       execve(...)                          |
          |                                 v
          v                             tsk2: child
     tsk1: mother                   ,--------------------,
,--------------------,              |  self_exec_id = 6  |
|  self_exec_id = 5  |              | parent_exec_id = 6 |
| parent_exec_id = 6 |              |    parent = tsk1   |
|   parent = <bash>  |              |   exit_signal =    |
|   exit_signal =    |              |           SIGSEGV  |
|           SIGCHLD  |              `--------------------`
`--------------------`                         |
          |                                    |
          v                                    v
      execve(SUID)                         exit(...)
          |                                    |
          v                                    |
     tsk1: mother                              |
,--------------------,                         |
|  self_exec_id = 6  |                         v
|   parent = <bash>  |                 x = tsk2->parent_exec_id = 6
|   exit_signal =    |                 y = tsk2->tsk1->self_exec_id = 6
|           SIGCHLD  |                      x == y
`--------------------`                         |
  ^   original                                 |
  | "exit_signal"                              |
  |    SIGSEGV                                 v
  ^------------------------< Deliver original "exit_signal" (SIGSEGV) >
```

Proof of Concept (PoC):

I was able to generate an integer overflow on tsk->parent->self_exec_id and successfully pass the verification in the "do_notify_parent" function. This allowed me to send an arbitrary signal to the mother process running in the different security domain. However, it takes relatively long time to generate

```
such integer overflow:

 1) test_case 1:
    Machine: VM under Hyper-V hypervisor
    CPU:     2 VCPUs (i7-8850H CPU @ 2.60GHz)
    RAM:     3.4 GB
    OS:      Ubuntu 19.10 (eoan)
    kernel:  5.5.1-050501-generic

    Time to overflow -> 17 days*

 2) test_case 2:
    Machine: Bare-metal
    CPU:     12 CPUs (Xeon(R) E-2176G CPU @ 3.70GHz)
    RAM:     32 GB
    OS:      Ubuntu 18.04.3 LTS (bionic)
    kernel:  4.15.0-72-generic

    Time to overflow -> 7 days*

*) However, my test cases were NOT optimized for speed and it is likely possible
   to reduce amount of time needed for such overflow.

After successful verification that described scenario is possible to achieve,
I've created a kernel module which simplifies my tests and instead of waiting
for a few days to generate an integer overflow, it simulates it. Kernel module
exports 2 IOCTLs:
 1) Dump current process':
    -> current process' task_struct pointer
    -> current process' PID
    -> current process' self_exec_id
    -> current process' parent_exec_id
    -> parent's task_struct pointer
    -> parent's PID
    -> parent's self_exec_id
    -> parent's parent_exec_id
 2) Overwrites parent's self_exec_id with value -100

I've adopted my PoC to leverage new kernel module and immediately simulate
described attack. Here are some notes from my tests:

 -> Ubuntu sets 2 as a default value for /proc/sys/fs/suid_dumpable
    -> This value requires that pattern set in /proc/sys/kernel/core_pattern
       must be either an absolute pathname (starting with a leading '/'
       character) or a pipe.
    -> Ubuntu is shipped with 'apport' package by default. This means that
       crashes are forwarded through pipe to the 'apport'
 -> However, apport throws an unhandled exception in case of ruid != uid, e.g.:

        ERROR: apport (pid 12773) Mon Mar  9 21:07:00 2020: called for pid 12762, signal 11, core limit 0, dump mode 2
        ERROR: apport (pid 12773) Mon Mar  9 21:07:00 2020: not creating core for pid with dump mode of 2
        ERROR: apport (pid 12773) Mon Mar  9 21:07:00 2020: Unhandled exception:
        Traceback (most recent call last):
          File "/usr/share/apport/apport", line 589, in <module>
            info.add_proc_info(proc_pid_fd=proc_pid_fd)
          File "/usr/lib/python3/dist-packages/apport/report.py", line 548, in add_proc_info
            self['ExecutablePath'] = os.readlink('exe', dir_fd=proc_pid_fd)
        PermissionError: [Errno 13] Permission denied: 'exe'
        ERROR: apport (pid 12773) Mon Mar  9 21:07:00 2020: pid: 12773, uid: 1000, gid: 1000, euid: 0, egid: 0
        ERROR: apport (pid 12773) Mon Mar  9 21:07:00 2020: environment: environ({})

    This situation is a results of dropped privileges by apport:

        euid = os.geteuid()
        egid = os.getegid()
        try:
            # Drop permissions temporarily to make sure that we don't
            # include information in the crash report that the user should
            # not be allowed to access.
            os.seteuid(os.getuid())
            os.setegid(os.getgid())
            info.add_proc_info(proc_pid_fd=proc_pid_fd)
        finally:
            os.seteuid(euid)
            os.setegid(egid)

 -> If you set 1 as a value for /proc/sys/fs/suid_dumpable (not recommended),
    you are free to fully play with crashdumps.


   Vectors of attack:

    The most obvious attack is to generate a coredump from the higher privileged
processes (like SUID binary). However, most of the modern distros should be
securely configured to protect from the attack using such situation (but I just
verified Ubuntu case).
Nevertheless, SIGSEGV is not the only useful signal to send. It is possible that
some privileged applications (SUID) might change their behavior based on the
signals which they receive. Various apps might implement signal handlers in
a various way which might be exploited using a described kernel bug. More
research is needed in that field.


   Details (Problem II.):

    Until 2012, 'self_exec_id' field (among others) was used to enforce
permissions checking restrictions for /proc/pid/{mem/maps/...} interface.
However, it was done poorly and serious security problem was reported known as
"Mempodipper" (CVE-2012-0056). More details about the bug can be found here:

https://git.zx2c4.com/CVE-2012-0056/about/

Linux kernel received a patch for that issue which completely changed the logic
how permission checks are enforced for /proc/pid/{mem/maps/...} interface. It
can be found here:

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e268337dfe26dfc7efd422a804dbb27977a3cccc

Since that patch, 'self_exec_id' is not tracked anymore, but kernel is looking
at process' VM during the time of the open(). However, as Solar Designer pointed
out, this logic might generate some problems:

https://www.openwall.com/lists/oss-security/2012/01/22/5

In short, if you hold the file descriptor open over an execve() (e.g. share it
with child) the old VM is preserved (refcounted) and might be never released.
Essentially, mother process' VM will be still in memory (and pointer to it is
valid) even if the mother process passed an execve().
This is some kind of 'memory leak' scenario. I did a simple test where process
open /proc/self/maps file and calls clone() with CLONE_FILES flag. Next mother
'overwrite' itself by executing SUID binary (doesn't need to be SUID), and child
was still able to use the original file descriptor - it's valid.

Nevertheless, I didn't explore that problem more. Maybe it is worth to do so?
Suspected resource limits bypass, to be confirmed or disproved with further
research.

Another interesting curiosity which is worth to point out about Problem II is
the Alan Cox's message sent during the discussion on the fix for CVE-2012-0056:

https://www.openwall.com/lists/kernel-hardening/2012/03/11/13


   Affected Software:

    Almost all Linux kernels should be affected. However, currently tested logic
is available since 3.3.5 up to the latest one (5.5.8).
Kernels 2.0.39 and 2.0.40 looks secure ;-)
```

```
Best regards,
Adam 'pi3' Zabrocki

--
pi3 (pi3ki31ny) - pi3 (at) itsec pl
http://pi3.com.pl
```