Kernel exploitation by example

Advanced Internet Security

Adrian Dabrowski
Christian Kudera
Georg Merzdovnik
Aljosha Judmayer

Kernel privilege escalation by example exploit

- `waitid()` CVE-2017-5123

```
Chris Salls discovered that when the waitid() syscall in Linux kernel
v4.13 was refactored, it accidentally stopped checking that the
incoming argument was pointing to userspace. This allowed local
attackers to write directly to kernel memory, which could lead to
privilege escalation.
```

- Introduced by commit 4c48abe91be03d191d0c20cc755877da2cb35622
- Fixed with commit 96ca579a1ecc943b75beba58bebb0356f6cc4b51

Background (CVE-2017-5123)

- `waitid()` syscall
- `int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);`

```
The waitid() system call (available since Linux 2.6.9) provides
more precise control over which child state changes to wait for.
The idtype and id arguments select the child(ren) to wait for, as follows:
idtype == P_PID
    Wait for the child whose process ID matches id.
idtype == P_PGID
    Wait for any child whose process group ID matches id.
idtype == P_ALL
    Wait for any child; id is ignored.
```

- `https://linux.die.net/man/2/waitid`

Background (CVE-2017-5123)

- The `siginfo` struct that gets written to the address pointed to by `infop`
- This structure usually gets written to userspace but since a check of the pointer is missing data in kernel space can be overwritten with this struct
- But because of the struct it not arbitary data can be written and the memory around the data that can be defined get mangled

```
struct siginfo {
    int si_signo;  // signal number
    int si_errno;  // always be equal to 0
    int si_code;   // signal code
    int padding;   // this remains unchanged/unused by waitid
    int pid;       // process id of child
                   // (somwhat controlled by forking but max value 0x8000)
    int uid;       // user id of user owning the process
    int status;    // return code 32bit,
                   // easiest to controll but constrained to be 0 < status < 256
}
```

Vulnerable kernel code (CVE-2017-5123)

- Last vulnerable version 4.13.6
  - https://elixir.bootlin.com/linux/v4.13.6/source/kernel/exit.c
- First fixed version 4.13.7
  - https://elixir.bootlin.com/linux/v4.13.7/source/kernel/exit.c

Vulnerable kernel code (CVE-2017-5123)

```
SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo __user *,
                                infop, int, options, struct rusage __user *, ru)
{
    struct rusage r;
    struct waitid_info info = {.status = 0};
    long err = kernel_waitid(which, upid, &info, options, ru ? &r : NULL);
    int signo = 0;

    if (err > 0) {
        signo = SIGCHLD;
        err = 0;
        if (ru && copy_to_user(ru, &r, sizeof(struct rusage)))
            return -EFAULT;
        }
        if (!infop)
            return err;

        user_access_begin();
        unsafe_put_user(signo, &infop->si_signo, Efault);    <-    no access_ok call
        unsafe_put_user(0, &infop->si_errno, Efault);
        unsafe_put_user(info.cause, &infop->si_code, Efault);
        unsafe_put_user(info.pid, &infop->si_pid, Efault);
        unsafe_put_user(info.uid, &infop->si_uid, Efault);
        unsafe_put_user(info.status, &infop->si_status, Efault);
        user_access_end();
        return err;
Efault:
        user_access_end();
        return -EFAULT;
}
```

Fixe kernel code (CVE-2017-5123)

```
SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo __user *,
        infop, int, options, struct rusage __user *, ru)
{
    struct rusage r;
    struct waitid_info info = {.status = 0};
    long err = kernel_waitid(which, upid, &info, options, ru ? &r : NULL);
    int signo = 0;

    if (err > 0) {
        signo = SIGCHLD;
        err = 0;
        if (ru && copy_to_user(ru, &r, sizeof(struct rusage)))
            return -EFAULT;
    }
    if (!infop)
        return err;

    if (!access_ok(VERIFY_WRITE, infop, sizeof(*infop)))
        goto Efault;

    user_access_begin();
    unsafe_put_user(signo, &infop->si_signo, Efault);
    unsafe_put_user(0, &infop->si_errno, Efault);
    unsafe_put_user((short)info.cause, &infop->si_code, Efault);
    unsafe_put_user(info.pid, &infop->si_pid, Efault);
    unsafe_put_user(info.uid, &infop->si_uid, Efault);
    unsafe_put_user(info.status, &infop->si_status, Efault);
    user_access_end();
    return err;
Efault:
    user_access_end();
    return -EFAULT;
}
```

Background (CVE-2017-5123)

- Kernel needs to be able to read and write memroy for the process which invoked a system call
- Therefore kernel has special functions like `copy_from_user`, `put_user` … which copy data from or to userland e.g., (on high level)

```
put_user(x, void __user *ptr)
    // checks that the ptr is in userland and not kernel memory!
    if (access_ok(VERIFY_WRITE, ptr, sizeof(*ptr)))
        return -EFAULT
    user_access_begin() // disable SMAP allowing the kernel to access userland
    *ptr = x
    user_access_end() // re-enable SMAP
```

- To avoid extra overhead of checks and enabling/disabling SMAP der are also unsafe versions: `__put_user`, `unsafe_put_user` without checks …

Escape the chrome sandbox (CVE-2017-5123)

- This vulnerability can be used in an privilege escalation exploit to break out of the Chrome sandbox.
- The Chrome sandbox should prevent compromise of the system in case of brower exploits.
- Therefore it restricts access to resources by:
  - changeing user id
  - doing a chroot
  - applying `seccomp` filter to block system calls that aren't needed.
- Normally very effective, but `waitid` syscall is usually allowed in `seccomp`.
- Note: Additional limitation imposed by sandbox is that no forking is allowd only threads, therefore waitid will always fail and only `0`s can be written to kernel memory.
  - Write only 24 bytes of `0`s and clobber nearby memory

Exploit strategy (CVE-2017-5123)

- Get an information leak and defeat KASLR
  - `unsafe_put_user` will not crash when accessing invalid memory but instead return `-EFAULT`
  - Therefore guessing where the kernel data and kernel heap section is possible because of different error messages

- Idea for this exploit:
  - Trigger actions that will result in the creation of “usefull structures” on the kernel heap (i.e., spray kernel heap), then try to hit them with arbitrary write of `0`s to escalate priviliges.

Exploit strategy (CVE-2017-5123)

- What action creates useful structres? -> create threads and overwrite their structure to reset the flag which marks if a `seccomp` filter is applied or not!
- `task_struct` represents each process and thread
  - https://elixir.bootlin.com/linux/v4.13.11/source/include/linux/sched
- Create 10000 threads that keep checking if they are still in the seccomp sandbox
- Stop if one finds that it is no longer affected by seccomp
- Then the exploit knows the address of its `task_struct` and there a task that is no longer under seccomp!
- Now the task can use `fork()` to create children and the exploit can write not only `0`s using `waitid()` status.

Exploit strategy (CVE-2017-5123)

- The author of the exploit used the `pid` field and shifts after every call to `waidid()` to create an arbitrary 5 bytes write.
- Now the author uses a technique from ret2dir
  - In `physmap` the kernel keeps aliases (second virtual address) mapped to the same physical memory as userspace memory.
  - By creating a page (acutally large amount of memory in userland) filled with some known value (`0x41`), this alias address that points to the exact same page can be found
  - Found by randomly overwrinting pages in the kernel `physmap` while checking if the userland page has changed.
  - If change is observed then the kernel virtual address that corresponds to this userland address has been found.
  - Now payload data can be constructed in userspace memory at a known kernel virtual address! => bypass SMAP

Exploit strategy (CVE-2017-5123)

- Now overwrite the `files` pointer in the known `task_struct` to point to the alias found in the kernel address space
- In aliased userland memory constrcut a fake `files_struct`, these objects have several function pointers e.g., `read`, `lseek`, `ioctl`
- Redirect function pointer to `ioctl` to ROP gadgets in the kernel (bypass SMEP) to get arbitrary read write.
- With that arbitary read write, first the clobbered portions of the `task_struct` have to be fixed!
- Finally remove the chroot by resetting the fs pointer
- Done …. fully escaped the Chrome sandbox!

Further resources (CVE-2017-5123)

- Blog post
  - https://salls.github.io/Linux-Kernel-CVE-2017-5123/
- Other exploits:
  - https://www.twistlock.com/labs-blog/escaping-docker-container-using-waitid-cve-2017-5123/
  - https://github.com/nongiach/CVE/tree/master/CVE-2017-5123