CVE-2021-30955 write-up

On February 23, @realBrightiup posted a write-up for the bug he had teased back in November. The write-up contained no PoC, or any code of any kind, so I decided to look into the bug and write one myself.

The bug can be found in the `ipc_kmsg_get_from_user` function in XNU 8019.41.5 (correspnding to 15.0.x) inside osfmk/ipc/ipc_kmsg.c. The function is called by `mach_msg_overwrite_trap`, when sending a mach message, and it is responsible for copying the message from userspace to the kernel. At first, it performs a few checks on the size argument, then it calculates this `len_copied` variable like so:

```
if (size == sizeof(mach_msg_user_header_t)) {
    len_copied = sizeof(mach_msg_user_header_t);
} else {
    len_copied = sizeof(mach_msg_user_base_t);
}
```

Here, `mach_msg_user_header_t` is just the barebones header (in which case, the message would be empty), whereas `mach_msg_user_base_t` is the header + 4 bytes, which is the minimum size a non-empty message can have. In the case of complex messages (messages which aside from raw data can also contain mach ports embedded in descriptor structures), those extra 4 bytes are taken by the `mach_msg_body_t` structure, which consists of a single field:

```
typedef struct{
      mach_msg_size_t msgh_descriptor_count;
} mach_msg_body_t;
```

This descriptor count specifies the number of descriptor structures we're going to send and the actual body data comes right after this field. In our case, we're interested on complex messages, so from now on assume said case. After this check, the function copies the base from userland like so:

```
if (copyinmsg(msg_addr, (char *)&user_base, len_copied)) {
    return MACH_SEND_INVALID_DATA;
}
```

Next, the descriptor count is saved in a variable:

```
descriptors = user_base.body.msgh_descriptor_count;
```

and then we reach this code (remember it!):

```
msg_addr += sizeof(user_base.header);
```

In there, msg_addr (the userspace address of the message) is changed to reflect for the fact that the header has been read, and that it's time to move on to reading the body data (as seen in the next few lines of code). Before reading the body though, it needs to allocate enough space to hold it, based on what it has read so far:

```
kmsg = ipc_kmsg_alloc(size, descriptors, IPC_KMSG_ALLOC_USER);
if (kmsg == IKM_NULL) {
    return MACH_SEND_NO_BUFFER;
}
```

The arguments passed here are the size of the message (adjusted for the differences in the size of the header between the kernel and userland) and the descriptor count from the body structure (0 in case of a simple message). The first few lines of ipc_kmsg_alloc are like so:

```
...
} else if (os_mul_overflow(user_descs, USER_DESC_MAX_DELTA, &max_delta)) {
    return IKM_NULL;
}

if (os_add3_overflow(size, MAX_TRAILER_SIZE, max_delta, &max_size)) {
    return IKM_NULL;
}
```

First, the descriptor count (user_descs) is multiplied by USER_DESC_MAX_DELTA, which is the difference of the sizes of the descriptor structures between the kernel and userland, in order to calculate the total difference in size, which is then stored in max_delta. Then, the size and max_delta are added together (along with MAX_TRAILER_SIZE, which is an irrelevant constant for us) to calculate the final size, used to allocate the data:

```
data = kalloc_data(max_size, alloc_flags);
```

The data and the size will then be stored in an ipc_kmsg struct (which is a kernel structure containing the message among other things), like so:

```
kmsg->ikm_size = max_size;
ikm_set_header(kmsg, data, size);
```

ikm_set_header then does this:

```
mach_msg_size_t mtsize = size + MAX_TRAILER_SIZE;
if (data) {
    kmsg->ikm_data = data;
    kmsg->ikm_header = (mach_msg_header_t *)((uintptr_t)data + kmsg->ikm_size
- mtsize);
} ...
```

If you do some calculations based on what we know about those values:

```
kmsg->ikm_header = data + kmsg->ikm_size - mtsize
= data + max_size - size - MAX_TRAILER_SIZE
= data + size + MAX_TRAILER_SIZE + max_delta - size - MAX_TRAILER_SIZE
= data + max_delta
= data + user_descs * USER_DESC_MAX_DELTA
```

So, `ikm_header` is set to our data allocation + the total difference in the sizes of the descriptor structures between the kernel and userland. This would leave an empty gap right before `ikm_header`, which might sound strange, but remember that in this stage the data is being copied as is from userland to kernel, without any adaptations being yet performed to make up for the different sized structures. As we'll see later, `ikm_header` will eventually be moved backwards to fill that gap, after the necessary changes are done on its data.

Going back to `ipc_kmsg_get_from_user`, after the message is allocated, the code sets up the header fields in the newly allocated `ikm_header` and then proceeds to read the message body and store it into `kmsg->ikm_header + 1` (so right after the header)

```
if (copyinmsg(msg_addr, (char *)(kmsg->ikm_header + 1),
    size - (mach_msg_size_t)sizeof(mach_msg_header_t))) {
    ipc_kmsg_free(kmsg);
    return MACH_SEND_INVALID_DATA;
}
```

Remember how `msg_addr` was changed by `sizeof(user_base.header)`? This second `copyinmsg` is reading the descriptor count all over again, and storing it in `ikm_header`. But what would happen if the descriptor count is changed during this time? The wrong count will be stored instead! This kind of race condition is known as a time-of-check time-of-use (TOCTOU) because the value of the variable being checked can be different during the check and during the use. In this case, we can use a different descriptor count to make the allocation, and a different one later on.

Now, we need to look at the code that adjusts `ikm_header` to remove the gap (remember, the gap = `user_descs * USER_DESC_MAX_DELTA`, which is just the number of descriptors times 4). The code is located in `ipc_kmsg_copyin_body`, precisely here:

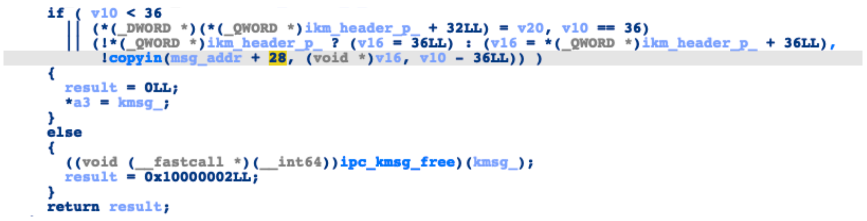
```
if (descriptor_size != KERNEL_DESC_SIZE * dsc_count) {
    vm_offset_t dsc_adjust = KERNEL_DESC_SIZE * dsc_count - descriptor_size;

    memmove((char *)(((vm_offset_t)kmsg->ikm_header) - dsc_adjust), kmsg-
>ikm_header, sizeof(mach_msg_base_t));
    kmsg->ikm_header = (mach_msg_header_t *)((vm_offset_t)kmsg->ikm_header -
dsc_adjust);

    /* Update the message size for the larger in-kernel representation */
    kmsg->ikm_header->msgh_size += (mach_msg_size_t)dsc_adjust;
}
```

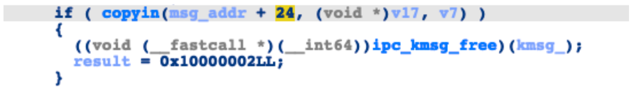
The code uses the descriptor count (now after being changed by the bug) to remove the gap before `ikm_header` by shifting all the data. One thing we could do to trigger a panic is initially set the descriptor count to 0 (which means no gap at all) and after allocation, we change it to a bigger value, which will then shift `ikm_header` to the left, landing out of bounds!

The patched code (starting from iOS 15.2) doesn't read the descriptor count again in the second copyin, so no TOCTOU is possible anymore (note 28 vs 24):



```
if ( v10 < 36
  || (*(_DWORD *)(*(_QWORD *)ikm_header_p_ + 32LL) = v20, v10 == 36)
  || (!*(_QWORD *)ikm_header_p_ ? (v16 = 36LL) : (v16 = *(_QWORD *)ikm_header_p_ + 36LL),
      !copyin(msg_addr + 28, (void *)v16, v10 - 36LL)) )
{
  result = 0LL;
  *a3 = kmsg_;
}
else
{
  ((void (__fastcall *)(__int64))ipc_kmsg_free)(kmsg_);
  result = 0x10000002LL;
}
return result;
```

Old code:



```
if ( copyin(msg_addr + 24, (void *)v17, v7) )
{
  ((void (__fastcall *)(__int64))ipc_kmsg_free)(kmsg_);
  result = 0x10000002LL;
}
```

The PoC (with comments):
https://gist.github.com/jakeajames/37f72c58c775bfbdda3aa9575149a8aa