How does one checkm8?

Introduction

This is my analysis and writeup of the vulnerabilities exploited in the checkm8 BootROM exploit. I wrote this in order to help me gain a better understanding of the vulnerability so that I could design my own strategy for exploitation and write my own implementation of the exploit. The checkm8 exploit relies on a couple of vulnerabilities:

- The main use-after-free (not patched until A14)
- The memory leak (patched in A12)
  The memory leak is essential in order to exploit the use-after-free because it allows us to deterministically craft the heap in order to allow the exploit to work. The patching of said memory leak in the A12 and A13 BootROMs is what prevents checkm8 from being exploited on these later SoCs.

Before we start, there are some important resources that I used to help me understand the exploit:

- This technical analysis of checkm8 by a1exdandy
- This presentation about checkra1n's implementation by Luca Todesco
- This vulnerability writeup by littlelailo
- This checkm8 "Q&A"
- ipwndfu by axi0mX
- gaster by 0x7FF
- The leaked iBoot/BootROM source codes - not linked here for obvious reasons
- securerom.fun for their collection of BootROM dumps that I reverse engineered

Throughout this writeup, any code examples will be taken from the pseudocode to show the flow of control, for legal reasons, but the corresponding functions can also be easily found within the leaked iBoot/BootROM source codes. Additionally, in order to simplify these examples, I have removed any unnecessary code and renamed variables to make them more readable. This includes various size checks and other safety checks that are not relevant. However, function names remain the same.

USB initialisation

USB is initialised within the `usb_init()` function, which will result in `usb_dfu_init()` being called. The function initialises an interface for DFU which will handle all USB transfers and requests. Furthermore, it allocates and zeroes out the global input/output buffer that is used for data transfers.

```
int usb_dfu_init()
{
  // Initialise and zero out the global IO buffer
  io_buffer = memalign(0x800, 0x40, 0);
  bzero(io_buffer, 0x800);

  // Initialise the global variables
  completionStatus = -1;
  totalReceived = 0;
  dfuDone = false;

  // Initialise the usb interface instance ... //

  return 0;
}
```

Information to take away from this:

- The global IO buffer, which holds all data from USB transfers, is allocated and zeroed-out
- Global variable to keep track of data received is initialised
- The global USB interface instance is initialised

Handling of USB transfers

When a USB control transfer is received by DFU, the `usb_core_handle_usb_control_receive()` is called. This function finds the registered interface for handling DFU requests and then calls the `handle_request()` function of that interface. In our case, this is the `handle_interface_request()` function, and the following code shows the control flow in the case of the host transferring data to the device. It checks whether the direction of the transfer is host-to-device or device-to-host, and then acts on the request in order to determine what to do next.

In the case of downloading data, which is what we will be using as part of the vulnerability, it will return one of three outcomes:

- 0 - the transfer is completed
- -1 - the wLength exceeds the size of the IO buffer
- wLength from the request - the device is ready to receive the data and is expecting wLength bytes

```
int handle_interface_request(struct usb_device_request *request, uint8_t **out_buffer)
{
  int ret = -1;

  // Host to device
  if ( (request->bmRequestType & 0x80) == 0)
  {
    switch(request->bRequest)
    {
      case 1: // DFU_DNLOAD
      {
        if(wLength > sizeof(*io_buffer)) {
          return -1;
        }

        *out_buffer = (uint8_t *)io_buffer; // Set out_buffer to point to IO buffer
        expecting = wLength;
        ret = wLength;
        break;
      }

      case 4: // DFU_CLR_STATUS
      case 6: // DFU_ABORT
      {
        totalReceived = 0;
        if(!dfuDone) {
          // Update global variables to abort DFU
          completionStatus = -1;
          dfuDone = true;
        }
        ret = 0;
        break;
      }
    }
    return ret;
  }
  return -1;
}
```

The important things to note from this are:

- The `out_buffer` pointer passed as an argument is updated to point to the global IO buffer
- It returns the wLength (provided it passes all the checks) as the length it is **expecting to receive** into the IO buffer

The result of this function, which was called from `usb_core_handle_usb_control_receive()`, is then used to indicate the status of the transfer, as shown below.

```
int ret = registeredInterfaces[interfaceNumber]->handleRequest(&setupRequest, &ep0DataPhaseBuffer);

// Host to device
if((setupRequest.bmRequestType & 0x80) == 0) {

  // Interface handler returned wLength of data, update global variables
  if (ret > 0) {
    ep0DataPhaseLength = ret;
    ep0DataPhaseInterfaceNumber = interfaceNumber;
    // Begin data phase
  }

  // Interface handler returned 0, transfer is complete
  else if (ret == 0) {
    usb_core_send_zlp();
    // Begin data phase
  }
}

// Device to host
else if((setupRequest.bmRequestType & 0x80) == 0x80) {
    // Begin data phase
}
```

As you can see, if the `handle_interface_request()` function returns a value that is greater than 0, the global variable for the size of the data expected to be transferred is then updated. It's also important to note that the `ep0DataPhaseBuffer` global variable will be updated to point to the global IO buffer if the device prepares for the data phas

This function is followed by the beginning of the data phase. The important parts of the function for handling the data phase are shown below, and the control flow of this function is crucial for understanding the main vulnerability here. After copying the data into the global data phase buffer, the function checks if all the data has been transferred. If so, it will reset the global variables in order to prepare for the next image to be downloaded.

```
void handle_ep0_data_phase(u_int8_t *rxBuffer, u_int32_t dataReceived, bool *dataPhase)
{
  // Copying received data into the data phase buffer
  // ...

  // All data has been received
  if(ep0DataPhaseReceived == ep0DataPhaseLength)
  {
    // Call the interface data phase callback and
    // send zero-length packet to signify end of transfer

    goto done; // Clear global state
  }
  return;
}
```

Once the data phase is complete, the data from the IO buffer is copied into the image buffer to be loaded and booted later on. After this, the following code is executed in order to clear the global variables as the data transfer is complete. This will then allow DFU to prepare to receive the next image over USB.

```
done:
  ep0DataPhaseReceived = 0;
  ep0DataPhaseLength = 0;
  ep0DataPhaseBuffer = NULL;
  ep0DataPhaseInterfaceNumber = -2;
```

This has been a lot to take in, so I will quickly summarise the process:

- In DFU initialisation, the IO buffer is allocated and zeroed out
- When transferring data, the global buffer for the data is set to point to the IO buffer
- Data transferred over USB is hence copied into the IO buffer
- When image transfer is complete, the contents of the IO buffer are copied into an image buffer
- This is followed by the resetting of the global state to prepare for a new image transfer

Use-after-free

Now, here's the fun part of the writeup - where I go into the actual vulnerability. When DFU mode is started, the main function that is called is the `getDFUImage()` function, the importants parts of which are shown below:

```
int getDFUImage(void* buf, int maxLength)
{
  // Update global variables with parameters
  imageBuffer = buf;
  imageBufferSize = maxLength;

  // Waits until DFU is finished
  while (!dfuDone) {
    event_wait(&dfuEvent);
  }

  // Shut down all USB operations
  usb_quiesce();
  return completionStatus;
}
```

So, what the function does is essentially allow for image transfers to happen and for DFU to do it's thing, and then shuts down the USB stack once it is finished. Now, looking back at the `handle_ep0_data_phase()` function, the global variables are all reset once the data phase has completed. However, if the data is *never fully transferred*, what happens then? The function simply returns **without clearing the global state**. This is good for us, as the attacker, because it means that the global variable holding the pointer to the IO buffer will still be intact.

Although it wasn't touched on above, taking another look at the `handle_interface_request()` function above will reveal that sending a `DFU_ABORT` command to DFU will cause it to set the `dfuDone` global variable to `true`, and signal the end of DFU. This can also be done by triggering a USB reset, which calls `handle_bus_reset()`. Back in `getDFUImage()`, this will result in the calling of `usb_quiesce()` to shut down the USB stack. The function looks like this:

```
void usb_quiesce()
{
  usb_core_stop();
  usb_free();
  usb_inited = false;
}
```

The `usb_free()` function calls `usb_dfu_exit()`, and the only important part of that function is the following:

```
if (io_buffer) {
  free(io_buffer);
  io_buffer = NULL;
}
```

So, let's summarise:

- Not completing the data phase results in the global variables not being cleared
- Sending a `DFU_ABORT` command results in the `dfuDone` global variable being set to true
- This causes `usb_quiesce()` to be called, leading to the IO buffer being freed
- The global variable pointing to the IO buffer remains, pointing at the now-freed buffer

As I'm sure you can now tell, this is the use-after-free utilised by checkm8. Next, I will go into how this vulnerability can be exploited, in order to gain code execution on the device. However, because the SecureROM is so deterministic, the IO buffer would normally be re-allocated over the freed one when the use-after-free is triggered - rendering the vulnerability useless.

This is also why the A12 and A13 SecureROMs are not vulnerable to the checkm8 exploit. They have the use-after-free, and it can be triggered, but there is no way to prevent the re-allocation of the IO buffer over the freed one.

So, to exploit this use-after-free vulnerability, a memory leak is utilised to manipulate the heap allocator into allocating the IO buffer at a different location on the heap during re-entry.

Memory leak

TODO

Exploitation

Unfortunately, to trigger the use-after-free with an incomplete data phase, you must go against the normal boundaries of USB transfers. There are two solutions for this that have been utilised in the open-source community: firstly, using micro-controllers (such as an Arduino + USB Host Controller) like this, to gain maximum control over the USB stack of the host device; secondly, forcing the cancellation of the transfer midway through, as is done in ipwndfu by using an extremely short timeout on an asynchronous transfer.

So, using what we know so far, the basic step-by-step for exploiting this vulnerability is as follows:

- Trick the heap allocator such that the new IO buffer will not be allocated over the freed one using the memory leak
- Send a setup packet with a DFU_DNLOAD request and a wLength within the correct boundaries to have the global variables set correctly
- Start a data phase transfer, but prevent it from completing, triggering the use-after-free
- Send a DFU_ABORT request or trigger a USB reset in order to free the IO buffer and cause DFU to re-enter
- Have DFU try to parse the image, but fail and then call getDFUImage() again using the original global variables
- Send and execute the exploit payload to give full code execution (more details to come)