Playing chess with a trillion-dollar company

Introduction

I'm no expert in hacking, but I really wanted to learn more about the iOS bootchain - so I decided to throw myself straight into the deep end and try to write a checkm8 implementation in C. I have included this writeup in order to document my journey, as well as help others who may be interested in doing something similar.

Now, a lot of what I've done in this project was very new to me when I started - namely:

- Reverse engineering
- Advanced C programming
- ARM assembly
- Exploitation in general
- Memory layout of the BootROM

For this reason, there is bound to be many mistakes or places where I could have written much better code. If you see anything that could be improved, please let me know and/or submit a pull request!

What is checkm8?

checkm8 is a BootROM exploit for Apple mobile devices with an A5-A11 SoC. It was discovered by axi0mX in the summer of 2018 and was released to the public in September 2019. It is a hardware-based exploit, meaning that it cannot be patched by Apple and affects all devices with the aforementioned SoCs.

The main vulnerability is a Use-after-Free in the DFU mode implementation of these SoCs. This Use-after-Free, along with an essential memory leak (that is the only reason this exploit works, and consequently doesn't work on A12 and A13) allows for arbitrary code execution in the BootROM.

The original exploit is a part of axi0mX's ipwndfu and was released along with the announcement on September 27th 2019. It was later ported to checkra1n by the checkra1n team, and is now used in said jailbreak as well as many other projects (such as palera1n).

With arbitrary code execution in the BootROM - at the lowest possible level on the device - one can have virtually-unlimited power over the device. It gives the ability to do things that normal, userland, jailbreaks don't allow for - such as dual-booting and using custom operating systems.

Setting up the project

I setup the original project in around May of 2023 - but I was in the middle of my GCSEs and so I didn't have much time to work on it. I setup a Makefile structure, organised the project and did some basic features such as logging and command line argument parsing.

For lack of a better name, I called the project "AlfieLoader" - named after myself. I had no intention of releasing this project to the public, and so I didn't really care about the name. I just wanted to get on with the project.

I began by setting up the exploit-related files. I wrote the code to find the device and connect to it, and handle errors such as no device. At this point, the program could find the device and connect to it.

I then setup a logging system - inspired by palera1n's logging system - in order to give colourfol log messages in the terminal. While I was doing this, I was using header files with `"../header.h"` and it got confusing - so I moved all header files to include so I could use `<header.h>` and make it cleaner.

In addition to this, I added a Makefile so all I have to do is type `make` and it will compile the project into `build/`. I also added a `make clean` command to clean up the project. I then build a basic command line argument parser, so I could pass arguments to the program (although there were few options - verbosity, debug mode, version and help menu).

This gave me the foundations to build the project on and saved a lot of time that wouldv'e been spent cleaning up the project later on.

Preliminary requirements for exploitation

First of all, I had to write the initial functions to find the device, connect to it and read it's serial number. You can see the relevant functions in `src/usb.c`. I then wrote some necessary code for the basic handling of DFU mode devices - such as parsing the serial number and created structures to hold the device information. This can be seen in `src/exploit/dfu.c`.

These DFU-specific structures were deprecated when I created the `device_t` structure (seen in `include/usb/device.h`) which could be used to handle devices in normal, recovery and DFU mode with a single structure type. After finishing the implementation of device classes, I decided to setup the actual USB transfer system. For this, I used a lot of the functions in gaster as it would have taken me too long to figure out how to get USB working correctly.

After this, it was time to begin the actual implementation of checkm8. I began by analysing the ARM assembly used by gaster and ipwndfu in order to get an idea of what the payloads do once executed on device. This was also helpful for my understanding of ARM assembly itself.

Becoming familiar with the checkm8 exploit

Writing up the exploit

Analysing implementations from ipwndfu and gaster

Creating my own exploit strategy

Writing my own implementation

Writing the payloads

Adding support for other devices

Conclusion

What's next?