AR.Pwn: Hacking the Parrot AR.Drone (Part 1)
Jeremy Rand
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy.rand@ou.edu

AR.Pwn: Hacking the Parrot AR.Drone (Part 1)

1 Introduction

It's been an exciting few years for Botball hackers. New hardware was introduced to the scene in 2011 (the AR.Drone), and again in 2013 (the KIPR Link). With all these new toys, the hackers have been going crazy, trying to figure out what to hack first. In 2012, AR.Drone libraries for the CBC were released by Jeremy Rand, Marty Rand, Kevin Cotrone, Ali Hajimirza, and Josh Maddux, but these libraries were more CBC hacking than AR.Drone hacking; the software running on the AR.Drone was untouched. This wasn't for lack of trying; it's simply that modifying the AR.Drone's firmware is very difficult. Meanwhile, the Link's 2013 release distracted everyone from the AR.Drone, but a few hackers quietly continued hacking away at the AR.Drone. The AR.Drone 2.0's release further complicated matters. We are now happy to announce that yes, the AR.Drone has finally been hacked. And we're going to tell you how.

But first, the obligatory disclaimer:

DISCLAIMER: Hacking the AR.Drone carries an inherent risk of bricking your AR.Drone. This risk is exacerbated if you mess with boot scripts or other sensitive components. We believe that our modifications are reasonably safe, and we have not experienced a brick, but we cannot guarantee that hacking your AR.Drone will not cause undesirable consequences. Hacking can void your Parrot warranty, and we cannot offer a replacement warranty. (However, please do notify us if something bad happens so that we can attempt to figure out what went wrong.) The AR.Drone firmware is copyrighted and is not open-source, meaning that distributing complete modified firmware images can violate Parrot's exclusive rights to distribute derivative works. Because of this, we do not distribute firmware images; we simply provide the means to modify your own AR.Drone. (There's no practical need to distribute your own images anyway... you'd probably just brick it.)

2 Hacking?

(Those of you who have read previous papers from the Botball hacking scene can skip this section, since presumably you already know what hacking is.)

Hacking is the use of clever technical tricks to modify a computer system to operate in a way which was unintended by the original creator, often involving some kind of reverse-engineering to figure out how the system works. Examples of hacking:

- Coding in C++, Java, or Lua in Botball.
- Wireless code downloading to the CBC.
- Playing Direct3D games in stereoscopic 3D or non-standard aspect ratios using specialized drivers.
- Playing Wii games with a VR glove or Kinect instead of a Wiimote.
- Playing LAN games over the Internet.

Hacking should not be confused with cracking, which is the the unauthorized bypass of security systems (usually for the purpose of obtaining confidential data or causing damage). Hackers generally don't like being conflated with crackers.

3 Background and Goals on AR.Drone Hacking

The Parrot AR.Drone is a consumer-oriented quadcopter aimed at augmented reality gaming. It uses WiFi connectivity to be remotely controlled (typically from an iOS or Android device using tilt control), and claims to use military-grade flight stabilization (including accelerometers, gyrometers, and camera tracking). At GCER 2011, Dr. David Miller announced that the AR.Drone would be used in a fall competition; this became the KIPR Autonomous Aerial Vehicle Contest, in which we placed first in December 2011. Our victory was mainly a result of custom libraries which we had developed.

We released CBC-compatible AR.Drone libraries at GCER 2012. These libraries were usable, but had a few shortcomings. Most notably, the cameras were extremely slow (around 1 fps, and around a 10-second latency), and a lot of navigation data (including altitude) was dropped. These shortcomings resulted in the widespread use of dead-reckoning as a replacement for both the camera and sonar, and We Deliver, the 2012 KIPR Open winners, did not use the AR.Drone at all.

Our primary goal, under the project name "AR.Pwn", was to improve the speed of both the cameras and sonar to the point that they were competitive with dead-reckoning.

Unfortunately, we have (so far) been so busy with the camera end of things that we have not yet had a chance to mess with the sonar. We also haven't yet had a chance to investigate the AR.Drone 2.0. If you're able to make progress in these areas, please let us know!

4 AR.Drone Internals

The AR.Drone runs Linux on-board. We were planning to run a port scan to see if it ran any interesting network services, but a quick Google later and we found that Kapejod.org had already done this and posted the results [1]. The AR.Drone is running a Telnet daemon as well as an FTP daemon. Telnet is a simple terminal protocol used for remote administration. The Telnet daemon will allow login as root (root is the standard admin account on Linux systems; the AR.Drone does not have a root password). FTP is the File Transfer Protocol, which allows read/write access to the `/update/` folder of the AR.Drone's filesystem.

The cameras are exposed as standard video4linux2 devices (`/dev/video0` and `/dev/video1`) according to Kapejod.org [1] (and verified by us). video4linux2 is the API standard to which all recent Linux webcam drivers conform. The navigation board, which handles accelerometer, gyrometer, and sonar sensors, is exposed as a serial port (`/dev/ttyPA2`) according to Kapejod.org [2]. All AR.Drone-specific code is in a binary file, `/bin/program.elf` according to Kapejod.org [1] (and verified by us). Killing `program.elf` will immediately shut off all motors and turn the LED's red. (Don't try this while your Drone is flying unless you like the idea of your Drone falling out of the sky like a rock.) `program.elf` requires exclusive access to the cameras and navigation board; it is only possible to interact with them while `program.elf` is killed.

All of our work was tested on AR.Drone firmware 1.7.6. We have no idea whether these hacks will work as intended on other firmware versions; do so at your own risk.

5 Running Without `program.elf`

Hugo Perquin has created an entirely homebrew flight controller for the AR.Drone [3], so `program.elf` is not necessary. However, Hugo's flight controller seemed to be less stable than `program.elf`, so we decided not to use it. We did find some major benefit from Hugo's code, though: he has full example code for accessing the AR.Drone's cameras as video4linux2 devices. This code helped us to understand how the camera devices worked, which enabled us to come up with some ideas for how we might be able to hijack them without killing `program.elf`.

6 Video Loopback Devices

Trying to have multiple programs access a Linux camera simultaneously isn't an unusual use case. In fact, there are multiple tools available, called video loopback devices, which are designed to allow multiple programs to share a single camera. The basic idea is that a simple proxy program reads the camera directly, and feeds the resulting video into several virtual camera devices, each of which can be locked by a different program.

We looked at several loopback device drivers. Unfortunately, none of them worked well with the Linux kernel version which the AR.Drone has on-board. Only one driver even built for the AR.Drone (we had to do significant hacking to even get it to build), and attempting to use it caused the Drone to reboot. (The documentation for this driver states that it has stability issues on the kernel which the AR.Drone uses.)

7 Shared Library Injection

After using a video4linux2 loopback device appeared to be a no-go, it was back to the drawing board. At Team SNARC, we're not particularly knowledgeable about hacking binary Linux programs, but we do watch hacking conferences. And conveniently, a recent talk by security researcher Dan Kaminsky at the 28th Chaos Communication Congress, called *Black Ops of TCP/IP 2011* [4], discusses injecting code into programs which use shared libraries, using a Linux feature called `LD_PRELOAD`. (This is probably common knowledge to hackers who are familiar with Linux... but Kaminsky's talk is the only time we've heard of it ourselves.) The specific uses that Kaminsky had for `LD_PRELOAD` are quite cool, but are outside the scope of this paper.

Shared libraries are a way to install a single instance of a library on a computer, such that every program which needs that library's features can dynamically link against that single instance. This reduces program executable size, and makes it easier to update libraries without having to update all the programs which use those libraries. In Windows, shared libraries have a `.dll` extension; in Linux, the extension is `.so`.

Under the hood, a shared library file contains a named list of available functions. A program linked against shared libraries contains a list of shared library names, and a list of functions which are to be accessed from those shared libraries. The operating system then imports those functions from the shared libraries into the program when the program is booted.

Linux has a feature called `LD_PRELOAD`. Take the following shell command:

```
LD_PRELOAD=./libinject.so ./myprogram
```

What does this do? Every function in `libinject.so` will override identically named functions in all shared libraries which `./myprogram` attempts to link with.

So how does this help us? Our goal is to obtain access to the hardware devices without killing program.elf. Well, all Linux programs interact with hardware devices using a few functions in a shared library, such as `open()`, `read()`, `mmap()`, and `ioctl()`. If we replace those functions, we could add our own implementation which interacts in some way with the data being exchanged with those devices. For example, we could dump data to a file, or replace the data before program.elf sees it.

8 How the AR.Drone Reads the Camera

The AR.Drone reads the camera according to the video4linux2 specification. Here's a rough outline of the process, based on Hugo Perquin's example code (and confirmed via experimentation with `program.elf`):

1. `program.elf` uses the `open()` syscall to gain exclusive access to the device.
    a. `open()` is passed the filename of the camera (`/dev/video0` or `/dev/video1`).
    b. `open()` returns a handle for the file.
    c. Our goal here is to hook `open()`, recognize that `/dev/video0` or `/dev/video1` is being accessed, and save the handle so that we can recognize future accesses to these devices.
2. `program.elf` uses the `QUERYBUF ioctl()` syscall to setup the buffers in the device driver.
    a. `QUERYBUF` is passed a file access handle (obtained from a previous call to `open()`), an index between 0 and 7 inclusive (8 buffers are cycled between) and an offset which is unique to each index.
    b. `QUERYBUF` returns no relevant information.
    c. Our goal here is to hook `QUERYBUF`, recognize that the file access handle matches the one we obtained in step 1, and save the combination of index and offset for later reference.
3. `program.elf` uses the `mmap()` syscall to map shared memory between the camera device and `program.elf`. This boosts performance since data doesn't need to be copied repeatedly.
    a. `mmap()` is passed a file access handle and an offset (corresponding to the offset we observed in step 2).
    b. `mmap()` returns a pointer to a video buffer.
    c. Our goal here is to hook `mmap()`, recognize that the file access handle matches the one we obtained in step 1, and save the pointer to the video buffer (indexed by offset) so that we can read the frame later.
4. `program.elf` uses the `DQBUF ioctl()` syscall to read a frame from the camera device.
    a. `DQBUF` is passed a file access handle and an index
    b. `DQBUF` returns no relevant information.
    c. Our goal here is to hook `DQBUF`, recognize that the file access handle matches the one we obtained in step 1, check the index of the frame being requested, check which offset we remember corresponds to the index, check where the buffer location is which corresponds to that offset, and finally gain access to that buffer

where the new frame lives.

9 Sharing Frames with Our Code

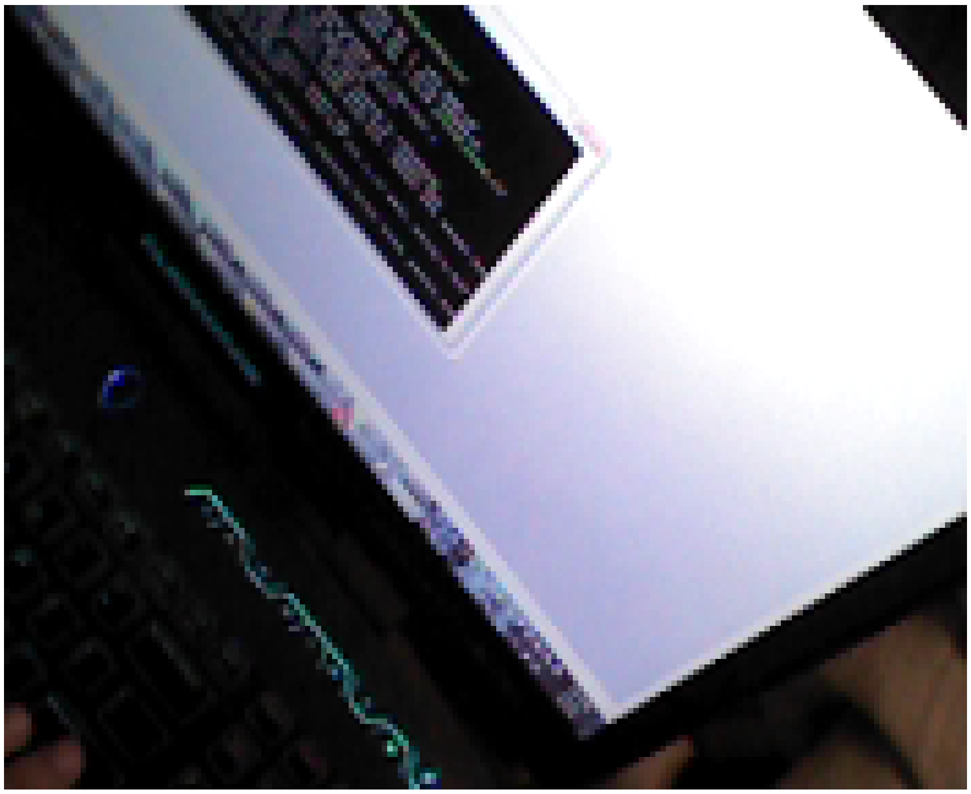
Once we have hooked the above syscalls using `LD_PRELOAD`, we instruct our hooked `DQBUF` function to copy the frame from the buffer into a file in `/tmp/`. We use `/tmp/` because it resides in RAM rather than flash. This improves speed, and also means that if we mess something up, we can simply reboot and the `/tmp/` folder will be wiped.

However, we have to be careful. If we simply dump to the same file every time the camera receives a frame, we may end up writing to the file while our soon-to-be-written vision code is in the middle of reading it. This would be a Bad Thing. If we instead write a new file every frame, we might quickly deplete the AR.Drone's RAM. This would also be a Bad Thing.

Instead, we use a flag file to allow the hook code and the vision program to arrange when to dump frames. The first time we dump a frame buffer, the hook code also creates an empty "ready flag" file. The vision program will wait for this flag to appear before trying to read a frame. When the vision program is finished reading a frame, it will delete the flag. Meanwhile, the hook will not dump any additional frames until it sees that the flag has been deleted; any frames which are received prior to that point will be skipped.

Below are the first ever raw images retrieved from the AR.Drone's cameras via AR.Pwn's hook.

Front camera: a wall and bookshelf (in Jeremy's bedroom).



Vertical camera: Jeremy's laptop, running a Telnet shell to the Drone

The lighting is dim in these images because these were the first frames accessed by the Drone during the boot process, before any kind of exposure control or gain was set to compensate for the dim lighting in Jeremy's bedroom in the middle of the night.

10 End of Part 1

This paper was too big to fit, so we've split it into two parts. Part 2 will continue where we left off here. See you in Part 2!

References

[1] kapejod. Parrot AR.Drone linux internals. http://www.kapejod.org/2010/08/parrot-ar-drone-linux-internals/ , 2010.
[2] kapejod. Let's set serial! http://www.kapejod.org/2010/10/lets-get-serial/ , 2010.
[3] Hugo Perquin. AR Drone program.elf Replacement. http://blog.perquin.com/blog/ar-drone-program-elf-replacement/ , 2011.

[4] Dan Kaminsky. *28c3: Black Ops of TCP/IP 2011*. https://www.youtube.com/watch?v=RifYnSKSkvk , 2011.