Vulnerability Analysis of AR.Drone 2.0, an Embedded Linux System

Ignacio Astaburuaga
Dept. of Computer Science & Engineering
University of Nevada, Reno
Reno, USA
Ignaciochg@nevada.unr.edu

Amee Lombardi
Damonte Ranch High School
Reno, USA
ALombardi@WashoeSchools.net

Brian La Torre
Sparks High School
Sparks, USA
BALatorre@WashoeSchools.net

Carolyn Hughes
Robert McQueen High School
Reno, USA
cjhughes@WashoeSchools.net

Shamik Sengupta
Dept. of Computer Science & Engineering
University of Nevada, Reno
Reno, USA
ssengupta@unr.edu

***Abstract*—The goal of this work was to identify and try to solve some of the vulnerabilities present in the AR Drone 2.0 by Parrot. The approach was to identify how the system worked, find and analyze vulnerabilities and flaws in the system as a whole and in the software, and find solutions to those problems. Analyzing the results of some tests showed that the system has an open WiFi network and the communication between the controller and the drone are unencrypted. Analyzing the Linux operating system that the drone uses, we see that "Pairing Mode" is the only way the system protects itself from unauthorized control. This is a feature that can be easily bypassed. Port scans reveal that the system has all the ports for its services open and exposed. This makes it susceptible to attacks like DoS and takeover. This research also focuses on some of the software vulnerabilities, such as Busybox that the drone runs. Lastly, this paper discuses some of the possible methods that can be used to secure the drone. These methods include securing the messages via SSH Tunnel, closing unused ports, and re-implementing the software used by the drone and the controller.**

***Index Terms*—AR, drone, 2.0, Parrot, security, embedded, Linux, Busybox, Vulnerabilities, CVE**

I. Introduction

Commercial Unmanned Aircraft Systems (UAS), often referred to as drones, are set to grow from 110,000 in 2017 to 450,000 in 2022 [1]. As their number grows so do their uses; for example, they can be used as bomb detection systems in the military, as first responders carrying a defibrillator, as tools for data collection in natural disasters such as wildfire mapping, payload delivery, disaster management, and rescue operations, as well as tools of surveillance by law enforcement [2]. In event of malfunction, like a malicious hack, the drone could be a safety hazard to the drone operator and any bystander, which could potentially injure someone. In some cases the drone carries a payload of some kind; for example, it could be a payload that could save someone's life or something the drone has to deliver like a very expensive asset. If the drone gets maliciously hacked, then it would not be able to carry out its duty of delivering the important payload, potentially causing hazard to an individual who needed help. The purpose of this research is to investigate and demonstrate the vulnerabilities present in the Parrot AR.Drone 2.0 so future drone manufacturers can build safer drones that are not a threat to public safety.

This document is a security analysis of the Parrot AR.Drone 2.0. It describes some of the possible attack vectors currently present in this legacy embedded system and some of the ways the system can be secured. This paper first discusses some terminology and inner workings of the drone. Then it goes into the steps taken to conduct this research and its results. Finally the paper discusses vulnerability analysis and some possible ways to secure the drone.

II. Theory & Definitions

The Parrot AR.Drone 2.0, which will be referred to as "drone" or "AR" in this paper, is an affordable and simple to operate drone that runs on outdated software. The drone runs a custom built Linux kernel, version 2.6, with multiple custom scripts, programs and services.

Parrot has dropped all support, development, and documentation for the AR 2.0, which made it difficult to conduct this research.

A. How the Drone Operates and Functions

The AR drone is operated almost like any other drone. The user controls the drone via a controller, which is a smartphone or tablet that runs the FreeFlight application (Android or iOS), rather than a dedicated hardware controller. The user connects his smartphone or tablet to an access point (AP) hosted by the drone and opens the FreeFlight application. At that point the user can operate the drone like any other drone on the market. The smartphone or tablet running the FreeFlight application will be referred to as "client" or "controller" in this paper.

B. Program.elf

Installed on the drone there is a proprietary software developed by Parrot called *program.elf*. This software is what interprets commands inside the drone. Once program.elf determines the meaning of the command, it is executed. If the command is hardware related, then it sends corresponding signals to the hardware controllers.

C. Busybox

Busybox is a single binary program that contains tiny versions of many UNIX utilities designed for small embedded systems [3]. In this research, the Busybox version found in the system was version 1.14.0, a lightweight custom build that had some UNIX tools removed. One of the removed tools is "passwd", a tool used to change or add a password to a user's account, allowing users to authenticate which increases security.

D. Pairing Mode

Pairing mode is a feature that restricts other controllers from controlling the drone, but it does not restrict access to its access point. It is the only security feature in the drone. It operates in the following way: the user connects his smartphone or tablet to the drone's access point, opens the application and enables the feature from within the settings. In the background the drone software writes down the remote controller's MAC address into the configuration file and runs "/bin/pairing_setup.sh". "/bin/pairing_setup.sh" uses *iptables* to block all traffic not coming from the MAC address of the now paired remote control. Section IV-C discuses how to circumvent the feature.

E. Communication

The controller and the drone interface with each other via multiple video, control, and configuration ports. The controller and the drone communicate using UDP packets; the drone receives the packets which are then interpreted by program.elf. Then program.elf communicates with the multiple motor controllers via serial communication [4]. There are also auxiliary ports for protocols like telnet and FTP, which are discussed in later sections of this paper. A basic diagram of the drone-controller interaction is shown in Fig. 1.

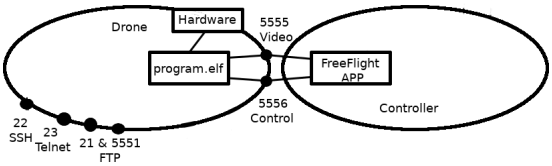

Fig. 1. This figure shows in a basic way how the drone communicates and operates. Connections outside of the controller or the drone are wireless routes, and connections inside are virtual routes.

AT* Commands, Communication Protocol:

The controller sends AT* commands to the drone, which ultimately control the drone. These AT* commands were originally detailed in the SDK provided by Parrot, but now this information is only accessible via third parties [5] [6].

These commands are sent to the drone via UDP packets. Program.elf only allows one control to be controlling the drone at a time, but since it communicates via UDP packets, if the pairing mode is disabled, the packets with the commands can be sent from any device, even if it is not a device that is running the FreeFlight application. If enabled, the paired controller's MAC address needs to be spoofed to be able to control the drone if control used is not the paired one. The commands have the following format:

AT*<command>=<sequence#>,<arg1>[,<arg2>]

Some of the most used commands are REF, PCMD, and CONFIG. REF is used for takeoff, landing, and emergency mode, PCMD is used for roll, pitch, and yaw, and CONFIG is used to configure some settings in the drone. The sequence number is a number that starts at one and is incremented by one. Program.elf does not interpret commands that have a smaller sequence number than previous commands. The only exception to this rule is if the sequence number is one, program.elf thinks the command is the start of a new connection.

More detailed information on the commands, commands not mentioned in this paper, and telemetry information can be found in the developers guide [6]. There is a library created by Felix Geisendorfer that supports the control, viewing of video streams, and viewing the navigation data of the drone through Node-js script [7].

III. Procedure, Vulnerability Discovery

A. Nmap Scan

An Nmap network scan was the first task done after connecting to the drone's network. The scan gave important details how the drone's system works. The scan revealed that multiple ports on the drone were open; results of the scan are shown in Table I. These ports have many different purposes, most of which are used by the FreeFlight application to control the drone. Some of the other ports do not control the drone but aid the controller in updating the drone. These ports include two FTP server ports and a telnet port. Anyone connected to the drone's access point has access to these ports. In an attempt to further understand how the drone-controller communication worked, a packet capture was carried out next.

TABLE I
Open Ports and Their Corresponding Description [8]

Port	Description
21 TCP	FTP: Serves videos and images recorded by drone
23 TCP	Telnet
5551 TCP	FTP: used for updates
5553 TCP	Video frames for client recording
5554 UDP	Navigation Data: telemetry
5555 TCP	Video stream
5556 UDP	Control interface: uses AT* commands to operate drone
5559 TCP	Configuration port: uses AT* commands

B. Wireshark Sniffing

Further information was obtained from doing a promiscuous packet capture, using Wireshark. The results of the packet capture are further discussed in the results section IV-A and in the communications section II-E. The next task done was to explore the operating system.

C. System Exploration Via Telnet

To explore the system even further, telnet was used, as it was shown in the Nmap scan as available with port 23 open. Once inside the system, it was discovered that the system runs Busybox for all of its UNIX commands and that scripts and files shown in Fig. II are the files that govern the drone by helping it initialize, configure, and later on control the system. At this stage it was discovered that the only security feature in the drone is the "Pairing Mode". The next task done was to explore some of the vulnerabilities present in the system.

D. Exploring Vulnerabilities on the Drone

Since there is only one security feature and the communication is in plain text, the research was focused on other vulnerabilities. Even though at the time of doing this research, there were 1,438 common vulnerabilities and exposures (CVE) for Linux kernel version 2.6, they were not explored [9]. Instead, vulnerabilities for the services running in the drone were explored. Busybox is the only program running in the drone (with the exception of scripts and program.elf), so it was researched next. Busybox 1.14.0 was found to have eight CVEs affecting it. Two of these CVEs will be explained more in detail in section IV-F. Exploit-DB and Metasploit were used to search for exploits that affected Busybox, but none were found. Metasploit had no exploits for the specific version, but it did have some generic information-gathering modules to scan the services. These modules were run against the drone's services and the results showed information already gathered, like Busybox version, etc. The next goal was to secure the drone.

Tools like Interactive Disassembler (IDA), Retargetable Decompiler (retdec), Radare2, and Snowman were used to reverse engineer program.elf to look for possible vulnerabilities. IDA and retdec did not work. Only Radare2 worked, but it disassembled program.elf to about 418,800 lines of assembly code making it difficult to reverse engineer. Later, Snowman was used to decompile it to C code. It decompiled program.elf to about 548,800 lines of code. At this point the idea of reverse engineering program.elf was abandoned.

E. Securing the Drone

The next step was to secure the drone's communication with an easy to implement method. This is where the idea of using an SSH tunnel was conceived, the results of which are further discussed in section V-E.

IV. Results & Vulnerability Analysis

The drone has many vulnerabilities, one of them being that all the ports are always open even when they should not be, e.g. telnet and FTP. There is no encryption on any of the messages that control the drone, including video feeds, telemetry, and configuration messages. There is one data validity check; therefore, data can be tampered with or fake packets can be crafted and sent. The only validity mechanism that the communication has is command sequence number. This sequence number is so program.elf knows when to discard older messages. This ensures that older commands that were delivered after newer ones are not interpreted or executed. With this in mind, anyone can send UDP packets to the drone and control it. The open access point contributes to the types of attacks that can be done. Pairing mode is the only security feature that prevents unauthorized control of the drone.

A. Communication & Packet Capture Results

The packet capture revealed that the communication from controller to drone and vice versa are in plain text. Because the communication is not encrypted or verified, the drone is susceptible to attacks like packet injection. In the event that a packet gets malformed or the sequence number is old, the packet is disregarded by program.elf.

B. Takeover & Deauthentication Attack

This drone is able to recover from a deauthentication attack after it has finished, but is unable to be operated while the attack is being carried out. If an attacker targets the original user's controller with a deauthentication attack, the attacker can connect his own controller and the drone will accept it as a new connection. Connecting to the drone will not work if pairing mode is enabled, since it drops all packets sent to the drone if they are not from the paired controller, but it can be circumvented. Commands can be sent directly to the drone and processed by program.elf. This DoS attack can be carried out using the command *aireplay-ng* from the *Aircrack-ng* suite. An Attacker can easily perform a device scan and look in the vendor section of the scanned MAC addresses for AR drones and attack them using the above mentioned methodology.

C. Circumventing Pairing Mode

There is only one security feature, and it can easily be bypassed in the event that the user enables it. The drone uses iptables to drop packets from devices that are not the paired controller. It can be bypassed by using Python library *Scapy*. This library can inject and spoof packets; in this case it can spoof the MAC address of the paired controller [10]. Packets can be spoofed to have the MAC address of the paired controller, bypassing the pairing feature. This also allows the attacker to craft a configuration packet that will tell the drone to disable the feature, but is not necessary to disable the feature to control the drone. The only two things that are necessary for the spoofing to work is to spoof the MAC of the controller and to either send a command with a sequence number higher than the last one, or send the command with the sequence number *one*.

TABLE II
FILES AND SCRIPTS WITHING THE DRONE

File	Description
/bin/program.elf	Main controlling software
/bin/check_update.sh	Checks for updates
/bin/init_gpios.sh	Initializes GPIOs to be able to communicate with hardware
/bin/mount_usb.sh	Mounts USB, mounts partition with the most available space
/bin/pairing_setup.sh	Setups paired mode (the only security feature)
/bin/parallel-stream.sh	Setups camera streams
/bin/reset_config.sh	Resets configuration file, keeping total flight time
/bin/umount_usb.sh	Unmounts USBs
/bin/wifi_setup.sh	Setups wiFi device and creates an AP
/data/config.ini	Configuration file for the drone, also contains information about the drone
/etc/inetd.conf	Configurations file for services, FTP USB and update folder setup
/etc/udhcpd.conf	DHCP server configuration file
/usr/sbin/loadAR6000.sh	Configures WiFi card

D. Telnet

By connecting to the drones network and establishing a telnet connection with the drone, a shell with root access can be obtained. Looking in the */etc/passwd* file shows that the root account has no password. This gives anyone full access to all the components of the drone. This includes access to binary, script, and configurations files, allowing us to edit the system. This also allows anyone to execute commands in the drone like *halt*, *reboot*, and *shutdown* which will make the drone completely stop its normal operation. If the drone is flying then it will drop from the sky potentially damaging it. This also allows access to execute other commands and scripts. A tool-chain can be used to cross compile C programs for the drone [11].

E. FTP

The drone hosts two FTP servers that are completely open. The FTP servers allow anonymous access with full read and write permissions. These two servers give access to the update folder */update/* and the media folder */data/video/*. USB storage devices are automatically mounted to the media folder, making accessible from the FTP server. The FTP servers were originally intended to be used to upload new firmware to the update folder and to download camera footage from the media folder. The FTP server can be used to facilitate the transfer of files to and from the drone.

The */etc/inetd.conf* file can be edited via telnet to host the entire filesystem and have full access to all the files via the FTP server.

F. Common Vulnerabilities and Exposures

Since all programs and services are part of Busybox, the only binary, program, and service (except for program.elf) that is exploitable in the drone is Busybox. The Busybox version running in the drone is version 1.14.0. By doing a search for vulnerabilities, exposures and exploits (CVEs) the following were found [12]: **CVE-2017-16544** shell autocomplete escape code execution; **CVE-2011-5325** Up to Busybox 1.22, tar implementation allows remote attackers to point to files outside the current working directory via system link; **CVE-2014-9645** Slash character can bypass some restrictions on loading kernel modules, Busybox <1.23.0, function add_probe in modutils/mosprobe.c; **CVE-2016-2147** DHCP client DHCPC in Busybox <1.25 allows a remote attacker to cause a denial of service attack(crash) via malformed RFC1035-encoded domain name, which triggers an out-of-bounds heap write; **CVE-2016-6301** The recv_and_process_client_pkt function in networking/ntpd.c in Busybox allows remote attackers to cause a denial of service (CPU and bandwidth consumption) via a forged NTP packet, which triggers a communication loop; **CVE-2016-2148** Busybox <1.25 Heap based buffer overflow(DHCP client udhcpc), allows remote hacker to have unspecified impact via vectors involving OPTIOM_6RD parsing; **CVE-2013-1813** util-linux/mdev.c in BusyBox before 1.21.0 uses 0777 permissions for parent directories when creating nested directories under /dev/, which allows local users to have unknown impact and attack vectors; **CVE-2011-2716** The DHCP client (udhcpc) in BusyBox before 1.20.0 allows remote DHCP servers to execute arbitrary commands via shell metacharacters in the HOST_NAME, DOMAIN_NAME, NIS_DOMAIN, and TFTP_SERVER_NAME host name options.

Only CVE-2017-16544 and CVE-2011-5325 out of all eight were explored further.

***CVE-2017-16544** shell autocomplete escape code execution*

Looking into CVE-2017-16544 shell autocomplete escape code execution shows that Busybox running in the drone is vulnerable. This vulnerability allows an attacker to execute escape code placed as a name for a file. Ariel Zelivansky shows the procedure to exploit this vulnerability, he uses C code to generate the file that will trigger the vulnerability [13]. He only explains in a simple manner how it can be applied to the Busybox system, while this research focuses on how these vulnerabilities apply to the drone.

The first step to be able to exploit this CVE is to create the file that has scape codes in its name, this is done with a C program that will not escape the scape codes. This step is important since most programs sanitize the file name by removing meaning from escape codes. A modified version of Ariel's code is shown in Fig. 2 and explained below.
The code does the following:

- `File` - not a command, text that gets printed. Later gets cleared.

```
#include <fcntl.h>
#include <sys/stat.h>
int main(int argc, char *argv[]){
  open(
    "file\e[2J\e[1;1H\e[47;5;32;15m
    ThisIsANormalFile\e[0m",
    O_RDWR | O_CREAT, S_IRUSR | S_IRGRP |
    S_IROTH);
}
```

Fig. 2. This figure shows a modified version of C program created by Ariel Zelivansky that creates a file with escape code in its name [13]. Note: there is not space in the string, it was broken up into two lines because of formating.

- `\e[2J` - clears screen, anything that was printed before will be pushed off the screen
- `\e[1;1H` - home-positioning cursor to 1,1. Moves cursor top left.
- `\e[47;5;32;15m` - pretty print any following text, until changed again
- `ThisIsANormalFile` - not a command, text that gets printed
- `\e[0m` - restore the original print attributes/style

It is important to execute this in the target drone. The C program has to be cross-compiled, this tutorial shows how to compile for the drone [11].

After compiling the file, it can be uploaded to the drone via FTP or USB. Then the binary can be executed to create a file containing escape codes in the name. Now a user can trigger trigger the execution of the escape code typing in a command that expects a file as argument like *cat*, pressing the *tab* key will show all available files which executes the code, the output should look like this:

```
$ cat <TAB>
ThisIsANormalFile
FileX
FileY
$
```

The attacker can also omit any visible characters so that the person who runs the code will not know that the attack was carried out. Some systems support screen capture and command executions via escape codes [14] but the AR.Drone 2.0 is not one of them since they are not implemented. This vulnerability is also present present in the command *ls*, which also executes the escape codes when printing file names.

***CVE-2011-5325** Tar Implementation*

This CVE is a bug in the implementation of tar program for this version of Busybox. Whenever the program *tar* untars a tar file that has a system link and a file that comes from a folder with the same name of the system link is extracted, the file from the folder gets extracted to wherever the system link pointed. This allows a user to put files in places where they shouldn't extract to. This vulnerability is present even when the -C flag that specifies a location. An attacker can get a privileged user to extract the specially crafted tar file, it can override files or even make new ones (e.g. a back door). All other tar implementations complain that this file cannot be untarred.

Denys Vlasenko shows in the documentation about the vulnerability the procedure to replicate the vulnerability [15]. He shows the relevant code that can be used to exploit the vulnerability. This research focuses on how to apply these concepts into the drone and see if these vulnerabilities are present in the drone.

The file that triggers the vulnerability can not be created in the drone since Busybox does not have the *append* command implemented, but it can be created in a Linux system and then transferred into the Busybox system. This means that the drone is vulnerable to this attack but it will require for the attacker to have access to the drone and place the tar file in it, then a user would have to extract it from within the drone to trigger the vulnerability.

In most cases this vulnerability could be used to escalate privileges by an unprivileged user. However this drone only allows the user root which means that this vulnerability does not make sense for this specific system.

V. Procedure, Securing the Drone

Securing the drone and its communication is an important aspect of security, which is why it was researched next. Multiple ways of securing the drone were tested without success. Only securing the communication via a WPA2 secured access point worked. Cross-compiling played an important role in updating some software.

A. Protecting from Pairing Mode Circumvention

The kernel can identify some types of packet spoofing through a method called *reverse path filtering* and it can be used by enabling hardware packet injection detection. In the case of the drone, the reverse path filtering did not want to stay enabled, so this approach of securing the drone was not successful.

B. Closing Ports

The drone always has all of its services running with their corresponding ports open, e.g. when the drone is flying the FTP and telnets ports are open, susceptible to attacks. A partial solution to securing the drone is to shutdown the services when not in use. This can be achieved by either stopping the services or by blocking the inbound ports. Iptables can be used to drop all packets for services that are not in use, and when in legitimate use allow packets.

Another potential solution would be to authenticate users when the services are needed to be used, but in this case the drone itself does not support the mechanism to authenticate users since it does not possess the capability to change users' passwords. This problem can be solved by updating Busybox to a more complete version that has such tools.

C. Hosting a WPA2 Secure Access Point

Securing the drone by securing the access point was the first idea explored. There are three possible ways to accomplish hosting a WPA2 secure access point.

1) Drone Hosting the AP: In order to host a secure AP, hostapd was cross compiled. Hostapd is an access point daemon that supports WPA2. After cross-compiling hostapd, it was tested, but it was not fully compatible with the hardware present in the drone. In the search for possible ways to make hostapd work, a thesis by Thomas Bertels was found. This thesis confirms that upgrading the security on the access point of the drone with hostapd can not be done due to compatibility issues [16], even when installing it through Ubuntu [17].

Connecting the drone to a WPA2 WiFi: The drone can connect to a open access point by reconfiguring the network interface as a client with the command *ifconfig* and *iwconfig* [18].

The drone can also be connected to a WPA2 access point by adding *wpa_supplicant* that Diego Araos compiled [19]. By using an access point with WPA2 the communication is secured.

2) Third Party Hosting the AP: The drone can be connected to a WPA2 secured access point that is not hosted by the drone or the controller. The controller has to be connected to the same access point. This method increases control range; the only drawback is that the connection is dependent on a third party to host the access point which at times it might not be practical.

In order for this method to work, the drone has to always be assigned IP address 192.168.1.1. This restriction happens because the FreeFlight application only searches for the drone under the IP 192.168.1.1, if it is not found there then it says the drone can not be found.

3) Controller Hosting the AP: A better solution would be to host the access point in the controller, via the hotspot feature that most phones support. In theory this approach should work, since all the controller is doing is sending UDP packets to the drone. For the same reason explained before, the FreeFlight application only looks for the drone in the IP 192.168.1.1 and it does not find the drone. This is also due to the fact that hotspots in mobile phones only support assigning IPs in the range of 192.168.1.0/24, and IPs can not be statically assigned.

D. Updating Busybox

Updates allows the removal of all the CVEs described in section IV-F, but it does not mean the system has none. The newer version may have vulnerabilities. A custom build of Busybox with removed commands is recommended in order to eliminate some potential vulnerabilities (if any) of applications or services that the drone do not use.

The update for Busybox is a simple process since all of the tools in the system are just system links to the single Busybox binary. The most recent version of Busybox was downloaded since a binary was available for the drone's architecture. Busybox can simply be updated by renaming the outdated Busybox binary, and renaming the new binary to what the old one was named. It is important to note that Busybox is the UNIX commands binaries in the system, so if it gets deleted then the system will not be usable anymore. When renaming the new version the *rename* command to move the new binary has to be called via Busybox directly e.g. "./busybox.old mv /pathA/busybox.new /bin/busybox". After the update all the commands are available, and systems links had to be created respectively.

E. SSH Tunnel

Another way to secure the drone would be to send all traffic through an SSH tunnel, while blocking outside access to other ports. An SSH tunnel is a method to connect or map two machines' ports via a secure SSH connection, it allows to map one machine's port to the other machines local ports. An SSH tunnel can be a regular tunnel like what is shown in or it can be a reverse tunnel, where a client connects to the server but the port mapping is done in the opposite direction, both shown on Fig. 3. Dropbear, an SSH server and client with key authentication support developed for POSIX based embedded systems [20], was a perfect fit; it compiled and ran, but due to compatibility issues with the outdated Linux system, it did not allow any logins to be made. Other options where explored, like installing other SSH server software. This presented the problem that there is no other reliable SSH server designed for embedded systems.

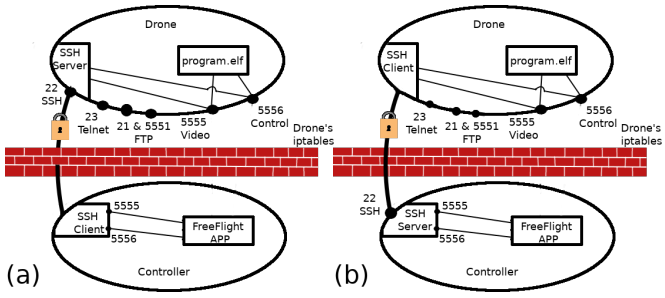

Fig. 3. This figure shows how (a) an SSH tunnel and (b) a reverse SSH tunnel can be used to secure the communication between the drone and the controller. If making a regulate SSH tunnel then an inbound Iptables rule would have to be made to allow SSH traffic at the drone's port. If using a reverse tunnel then the drone can block all incomming traffic exept for a port for drone management.

Installing Ubuntu 12.04: A simple solution to the problem is to run Ubuntu 12.04 on the drone via a USB drive [17]. Allowing to install any program from the Ubuntu repository, increasing the memory by adding more swap space, and running bigger programs not intended for an embedded system. The creation of the Ubuntu USB had to be done in virtual machine running Ubuntu 12.04 since using Ubuntu 17.10 was too new and some of the components of the installation process were deprecated.

After OpenSSH was successfully installed, an SSH connection could be made to the drone as well as SSH tunnels. There are two ways the tunnel can be hosted.

The first way is using the drone to host the SSH server, this way the controller connects and creates two regular tunnels,

one for the video port and one for the UDP control port. To secure the drone, iptables has to block all ports except the SSH port. With the regular SSH tunnel in place the FreeFlight application has to connect to *localhost:PORT*, where PORT is a port from Table I. The FreeFlight has to connect to a different location that is not 192.169.1.1, therefore this approach fails.

The second way to setup the SSH tunnels is to have the controller host the SSH server and then have the drone create both SSH tunnels. In this case the drone can close all the ports except the SSH port (it is not advised close the SSH since one would lose access to the drone). This approach also does not work because the FreeFlight does not allow the user to change the destination, and because there is no application that can host an SSH server that supports an SSH tunnel (at least on Android).

Although a connection could be made, this method did not work since the controller packets can not be easily routed through the tunnel. In theory this approach can work as long as the latency created by the tunnel is small and the client can connect to a different IP.

The next idea was to install OpenVPN on the drone and have it host a VPN. The controller would then connect to the VPN and send all the traffic over the encrypted VPN tunnel. This method did not work because the OpenVPN service did not start due to compatibility issues with the outdated Linux kernel.

VI. Conclusion & Future Work

The Parrot AR.Drone 2.0 is a legacy system that has multiple vulnerabilities, making it a possible target for a wide range of attacks. These include attacks like taking over the drone, controlling the drone, and full control of its services including full root access to its operating system. The best way to update this legacy system is to update it via telnet and FTP. It can be done by updating Busybox, implementing a new version of program.elf and the controller application that implements security features, and blocking ports when not used. Some of these security features can include encrypting the communication and having the controller and the drone verify the sender and the integrity of the message. Having something like a secure rolling code could help mitigate some of the problems present with the system. Hugo re-implemented program.elf and made a desktop program to interface with it, which can be used as a foundation to implement some encryption and data verification mechanisms [21]. Thomas Bertels' thesis goes into detail about methods of securing two way communication for drones. He and his team cover pairing protocols and encryption mechanisms for the drone [16].

Navigation data was not the focus of this research, but it is also an important aspect of security since it can leak information about the drone and the user. Shared library injection to get navigation data and video feed directly from the hardware without killing the program.elf [22] [23] and GPS hacking are other ways that this system is also vulnerable.

Acknowledgment

This research is supported by NSF Award #1542465.

References

[1] Faa.gov. "Fact Sheet Federal Aviation Administration (FAA) Forecast Fiscal Years (FY) 2017-2038". [Online] Available at: https://www.faa.gov/news/fact_sheets/news_story.cfm?newsId=22594 [Accessed: 7-Sep-2018].

[2] Brown, J. "Drone Uses: The Awesome Benefits of Drone Technology". [online] My Drone Lab. Available at: http://mydronelab.com/blog/drone-uses.html [Accessed: 7-Sep-2018].

[3] "BusyBox", Busybox.net. [Online]. Available: https://busybox.net/about.html. [Accessed: 10-Aug-2018].

[4] "AR Drone Motor Controller Tech Toy Hacks", Blog.perquin.com, 2011. [Online]. Available: http://blog.perquin.com/blog/ardrone-motor-controller/. [Accessed: 10-Aug-2018].

[5] "AR Drone 2/AT Commands - PaparazziUAV", Wiki.paparazziuav.org, 2017. [Online]. Available: https://wiki.paparazziuav.org/wiki/AR_Drone_2/AT_Commands. [Accessed: 10-Aug-2018].

[6] "AR.Drone Developer Guide SDK 2.0", Jpchanson.github.io. [Online]. Available: https://jpchanson.github.io/ARdrone/ParrotDevGuide.pdf. [Accessed: 10-Aug-2018].

[7] F. Geisendorfer, "felixge/node-ar-drone", GitHub. [Online]. Available: https://github.com/felixge/node-ar-drone. [Accessed: 10-Aug-2018].

[8] J. Pleban, R. Band and R. Creutzburg, "Hacking and securing the AR.Drone 2.0 quadcopter - Investigations for improving the security of a toy", in Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications, San Francisco (CA), 2014.

[9] "NVD - Results", Nvd.nist.gov. [Online]. Available: https://nvd.nist.gov/vuln/search/results?adv_search=true&cpe=cpe%3a%2fo%3alinux%3alinux_kernel%3a2.6.0&startIndex=0. [Accessed: 10-Aug-2018].

[10] P. Biondi and S. Community, "Scapy", Scapy.net. [Online]. Available: https://scapy.net/. [Accessed: 10-Aug-2018].

[11] H. Nadeem, "How to Cross Compile C/C++ application for AR Drone Hassan Nadeem", Hassan Nadeem, 2015. [Online]. Available: https://hassannadeem.com/blog/2015/04/28/how-to-cross-compile-for-ar-drone/. [Accessed: 10-Aug-2018].

[12] "NVD - Results", Nvd.nist.gov. [Online]. Available: https://nvd.nist.gov/vuln/search/results?adv_search=true&cves=on&cpe_version=cpe%3a%2fa%3abusybox%3abusybox%3a1.4.0. [Accessed: 10-Aug-2018].

[13] A. Zelivansky, "Busybox Shell Vulnerability CVE-2017-16544 Twistlock Alerts", Twistlock, 2017. [Online]. Available: https://www.twistlock.com/2017/11/20/cve-2017-16544-busybox-autocompletion-vulnerability/. [Accessed: 10- Aug- 2018].

[14] D. Lukan, "A Blast From the Past: Executing Code in Terminal Emulators via Escape Sequences - Protean Security", Protean Security, 2014. [Online]. Available: https://www.proteansec.com/linux/blast-past-executing-code-terminal-emulators-via-escape-sequences/. [Accessed: 10-Aug-2018].

[15] D. Vlasenko, "busybox - BusyBox: The Swiss Army Knife of Embedded Linux", Git.busybox.net, 2011. [Online]. Available: https://git.busybox.net/busybox/commit/?id=a116552869db5e7793ae10968eb3c962c69b3d8c. [Accessed: 10-Aug-2018].

[16] T. Bertels, "Design of a pairing protocol for the AR.Drone 2.0", KU Leuven, Technology Campus De Nayer, 2016.

[17] "Ubuntu on AR.Drone 2.0 - drone-forum.com", Drone-forum.com, 2013. [Online]. Available: https://www.drone-forum.com/forum/viewtopic.php?t=7094#p71757. [Accessed: 10-Aug-2018].

[18] M. Monajjemi, "AutonomyLab/ardrone_autonomy", GitHub, 2013. [Online]. Available: https://github.com/AutonomyLab/ardrone_autonomy/wiki/Multiple-AR-Drones. [Accessed: 10-Aug-2018].

[19] D. Araos, "daraosn/ardrone-wpa2", GitHub, 2013. [Online]. Available: https://github.com/daraosn/ardrone-wpa2. [Accessed: 10-Aug-2018].

[20] "Dropbear SSH", Matt.ucc.asn.au. [Online]. Available: https://matt.ucc.asn.au/dropbear/dropbear.html. [Accessed: 10-Aug-2018].

[21] "AR Drone program.elf Replacement Tech Toy Hacks", Blog.perquin.com, 2011. [Online]. Available: http://blog.perquin.com/blog/ar-drone-program-elf-replacement/. [Accessed: 10-Aug-2018].

[22] J. Rand, "AR.Pwn: Hacking the Parrot AR.Drone (Part 1)", Files.kipr.org. [Online]. Available: http://files.kipr.org/gcer/2013/proceedings/Rand_Hacking_AR_Drone_1.pdf. [Accessed: 10-Aug-2018].

[23] J. Rand, "AR.Pwn: Hacking the Parrot AR.Drone (Part 2)", Files.kipr.org. [Online]. Available: http://files.kipr.org/gcer/2013/proceedings/Rand_Hacking_AR_Drone_2.pdf. [Accessed: 10-Aug-2018].