Christopher Akatsuka, Sagar Sinha, Jeremy Danziger

A Discussion of AR Drone Parrot Security

Introduction

The Parrot AR Drone 2.0 is a drone quadcopter with the capabilities for 2 angles of video feeds, as well as significant flight capabilities. However, this device contains a number of its vulnerabilities in its design which leave open the opportunity for malicious attacks which could threaten the confidentiality, availability, and integrity of the usage by the drone and the drone's controller. The following paper will discuss some of these vulnerabilities, and how they were exposed. Some of the major vulnerabilities include being able to capture video from the drone, being able to send commands to the drone, and being able to act as a Man in the Middle between the drone and the device itself. After reviewing the vulnerabilities we found, we will then discuss measures that can be taken to improve the drone's security to prevent such attacks from happening.

Issues and Exploits

The first and most obvious security issue is the WiFi connection itself. In order to connect to an AR Drone from a device, one need only connect to the wifi network of the device. Our attack to overcome this was not much of an "attack", as we simply connected to the WiFi on our laptop, and, using packet capture methods, were able to gather a large amount of information about packets which were being sent across the network. This allowed us to gather a significant amount of information on how the

drone's operation worked with regard to communication between the phone and the drone itself, as well as begin to give us insight one what filters we would need to apply in order to properly capture the correct packets for our purpose.

For Part 1, we discovered that due to the process and data-type through which video is sent from the drone to the phone controller, it was possible to intercept these packets using tcpdump in monitor mode, and assemble the packets using scapy's sniff function in order to create a decent recreation of the video feed, which could be viewed in ffplay. Through the user manual for the AR Drone 2.0, we were able to quickly understand how the video feed worked (multiple packets with timestamps to identify the packets), what packets were used (TCP), and the data-type and compression used (h264 with custom headers). Armed with this information, we used python, tcpdump, and ffplay to create an exploit which captured video from the drone. This exploit worked as follows.

1. Clear out any pre-existing files from an older run of the program.
2. Set the channel to listen on (the drone's network appeared to statically be on channel 6).
3. Create a general tcpdump file to store all packets captured over the time period of the capture.
4. Over the course of each second, grab packets in a separate tcpdump call, filter out non-video packets, and append the video packets to an ongoing stream being played on ffplay (Video packets originated from the Drone's IP and were sent from port 5555).

    a. Play this stream as soon as a frame is received
5. Upon user termination, stop the program, kill any remaining processes, and clean up temporary files

Therefore, as you can see, the method for grabbing packets was relatively straight-forward. We chose to use ffplay because it has built in capabilities to strip custom headers (such as the ar-drone's PaVE header) from the ar-drone files, thus allowing frames to be easily added once rebuilt, as no major processing was required in order to play the video from the stream coming from one of its two cameras. tcpdump helps with useful packet capturing and filtering purposes. The scapy module was extremely helpful when it came to sniffing old packet capture files, as the built-in parsing for scapy allows for each manipulation of the contents of any given packet. Interestingly, older versions of the drone used a proprietary video format, rather than the more common h264 for this version. This may have been more difficult to crack - however, the SDK still included details about the implementation, so the older SDK could have still been managed with a bit of extra work

For part 2, we set out to expand upon our part 1 exploit. That is, rather than just view what was going on in the drone's video feed, we sought to take over the drone, and issue our own commands to the drone. Our exploit worked in the following way:

1. Identify the MAC Address(es) of the device(s) currently controlling the drone.
2. Telnet onto the drone's network, 192.168.1.1
3. Add to its filter to ban the mac address of the device which is currently being used to control, which is generally 192.168.1.2. If necessary, block all MAC

addresses that are not the attack laptop, until the attack laptop is the primary user

    a. Leverage the programming mechanism which switches the connections for both video and control to the next lowest IP address, which is generally 192.168.1.3 (ie the attacking computer)

4. Wait until the controller loses both video and control feed, and thus the drone finds a new controller to send packets to.
5. Using one of the many libraries available, issue commands to the drone via the laptop.
6. Execute desired commands for controlling the drone through the library.
7. Enjoy new Drone.

This exploit was relatively straightforward. For it, we simply needed to figure out who else was on, and kick them off. Determining who else was on and which device was currently connected was a simple matter of running ‘netstat’ and ‘arp -a’. Once we realized we could map this out, we even scripted this mapping into our exploit so you don’t even have to physically telnet onto the drone. Kicking off of the network was incredibly easy due to the lack of restrictions on modifying iptables after telnetting onto the drone. By taking advantage of the drone’s programming of going to the next IP to determine who is under control, we simply needed to put ourselves in the position to take control, and make sure that no one else one the network could try and stop us from taking control.

On github, due to a large number of developer's desires to use the drone for open-source development (particularly for object recognition/tracking and other machine learning applications), there were a large number of libraries which provided the control commands which would automatically construct and relay packets. This is for a couple of reasons. For one, the layout of how navigation work is laid out in the SDK. Furthermore, the commands for movement are sent via UDP, meaning anyone can send such data without establishing a "streamed" connection. Therefore, even beyond SDK's, injection of navigation packets are possible. Lastly, the commands for navigation essentially boil down to a handful numbers which correspond to possible axes of movement for the drone, along with the take off, landing, and status commands, for which the packet's payload looks almost like a dictionary. In summary, the commands for controlling the drone are not only easily delivered, but also easily created and understood.

Using the node library ar-drone, we were quickly able to issue commands to the drone as its new primary user. A script was then created in order to assist us in controlling the drone via the keyboard. Together, this attack can be executed in a matter of seconds, quickly allowing for an attacker to take over the drone.

Other Vulnerabilities

These drones are vulnerable to a few other attacks. For instance, as was accidentally done at points during attempting a Man in the Middle (MitM) attack, it is possible to perform some version of a Denial of Service attack. This can be done by

performing a MitM attack, but not forwarding packets, thus preventing the exchange of information between the drone and the phone controller. Alternatively, one can flood the network with ARP packets as well. This can be used to temporarily interfere with drone activity, though it may not produce some immediate reward for an attacker. However, if the goal is to simply disable drone actions, then this attack could be appealing.

Similarly, as mentioned, the network is susceptible to a MitM attack. It is relatively straightforward to put an attacking device as an in-between for the drone and the phone controller. While in the middle, an attacker could easily scrape information, or maliciously change packet payloads between the two devices. This includes sending commands which may cause the controller to lose control of the drone or performing commands that the controller did not intend. Alternatively, it could be used to provide a fake video feed to the user, causing a user to mishandle the drone in some way.

This could be done via a python script which issues ARP commands to both end points claiming to be the the other using scapy. Through this, one could filter through all the traffic that goes between the two, and be undetected by either endpoint via IP forwarding all the while sending erroneous video packets to the phone. The README.txt file in the EC directory contains more detailed information about how we implemented our MITM attack.

Defenses

1. Telnet access onto the drone should be disabled (or at least add a password). Allowing telnet access enabled us to take complete control over the drone (primarily by locking out actual users from connecting to the drone's network with

the use of iptables) without any barriers to entry. We literally only had to connect to the drone's network and we could telnet on. Without telnet access, our attacks would have been much harder to execute.

2. In order to protect the drone from even naive connections, the drone should implement some sort of WiFi security measure, to make it far more difficult for an attacker to connect to the Drone. Some discussion online seem to indicate that MAC filtering is possible to add security, but adding secure WiFi would be far more practical as well as accessible for a general user. Therefore, simply adding secure WiFi could essentially help prevent security issues downstream for the drone.
    a. We recommend using the WPA2-PSK(AES) encryption. The drone will be configured as a WLAN node with a passphrase. Any traffic sent to the drone will only be accepted if the client provides the correct password. Thus, any spoof command packets sent by the attacker imitating the controller, will be rejected by the drone. Using encryption will further ensure the protection of the TELNET and the FTP services provided by the drone. Only the controller will be able to establish any form of communication. The protections of these services are essential since the attacker can use it to figure out the controller's identification by telnetting into the drone, disconnect any connections, and upload malicious files on the drone which can possibly render the drone useless.

3. In addition to encrypting the network, we also suggest encrypting the payload of packets. This way, it will be much more difficult to sniff and reproduce what the controller is seeing. Moreover, information such as commands issued and drone state can also be protected.
    a. While in some cases secret key encryption on the payload is not possible, agreeing on a secret key in this case is possible as the two devices involved in the transfer of information are forced to be in close proximity to each other, and in general tend to have the same user. However, the encryption must be done carefully. As it turns out, the video section of the data packets does have a header. The presence of a header gives some information, for one, and unfortunately the information contained within the header is generally guessable, which could enable attackers to easily decrypt the video stream. Therefore, any encryption created for the data packets must only encrypt the non-header video frame, which is likely more difficult to predict and crack. Using encryption, streams such as the one used in the exploit will become significantly harder to use.
4. Any device can seem to arbitrarily issue commands to the drone if its ip address is the device bound to the right ports. Thus, if we enabled MAC filtering to only allow in the mac addresses of approved devices (ie whitelist owner's devices and blacklist all others), we can at least prevent some basic network attacks.
5. The commands to the drone to control its actions are sent using UDP. We recommend using TCP instead. The drone only checks if the sequence number

specified in the AT command is greater than the one it previously received. This makes it hard to detect loss of packets due to a denial of service. Using TCP also means it will be hard to hijack a connection by guessing the TCP sequence number, though not impossible.

6. Logging all the activities on the drone can give useful insights into possible attacks on the drone. The operator can then set up some sort of detection and action mapping.
7. Provisioning of a system where the drone's software can be patched. Since the drone cannot and should not connect to the Internet, the patches will have to be installed by the controller. The drone should only install patches provided with digital signatures from Parrot. Having regular updates will ensure the drones which have been dispatched are protected against known vulnerabilities.
8. Having a restriction on the controller's access of the drone's file system. We believe that the controller should only have user level capabilities instead of root level capabilities. The current ability of the user to modify the iptables essentially enables any laptop to knock off other devices for which they have the MAC address.
9. Using hardware encryption to protect the integrity of the drone's Read Only Memory. This will ensure that the ROM is not tampered with or read by the attacker.
10. If the intention of the drone manufacturer is to safely allow more than one controller to use the drone at different times then in addition to passwords,

usernames will be required. Further, a user should not be able to access files of other users. This is in regard to the creation of recorded video and pictures taken by the drone during flight.

11. Finally, although it sounds strange, releasing the SDK for the drone introduced some security vulnerabilities. Having access to a software development manual made it far easier to understand the infrastructure of these machines. While the lack of a manual certainly would not have hindered development, it certainly would have slowed us down a bit.
    a. Still, given the benevolent applications of most libraries, it is tricky to weigh popularity with software security. It is more likely than not that attackers could sufficiently crack the drone without needing the SDK. Therefore, the commercial and popularity benefits of having it likely outway the risk of releasing it in the eyes of the developers.

Conclusion

In conclusion, the AR Drone 2.0 contains a number of security issues which we have exploited, and for which Parrot should consider trying to address. Some of these issues, such as protecting the WiFi, are even straightforward, and yet the lack of them creates a hole which our attacks can easily take advantage of. Without further improving security on future Parrot products, the company threatens the confidentiality of user's actions (and potentially their information), the integrity of the data exchanged between the two devices, and the continued availability of being able to use such a device. Introducing further countermeasures could not only benefit users, but also potentially

increase the popularity and thus revenue of the Parrot company, at the cost of only a bit more time invested into better protecting their devices.

Attributions

Please see each directory's individual README for a detailed description of software packages used, how to operate the code, and lists of sources we leveraged to implement our attacks.