6.858 Final Project

Predrag Gruevski
Paul Hemberger
Andres Romero
Albert Wu

Introduction

Since remote desktop applications are such powerful tools for controlling computers from across the Internet, it stands to reason that adversaries would be motivated to discover and exploit vulnerabilities in such software. In our final project, we explored many of the most popular remote desktop applications for Android clients. Most of them provide little to no security, and we were able to easily create exploits to compromise the keyboard and mouse of the remote machine.

Methods

We looked at the following popular Android clients (each has at least 100,000 installs on the Google Play app market):

1. TeamViewer for Remote Control
2. Air HID
3. Remote Mouse
4. WiFi Mouse
5. Android Mouse and Keyboard
6. Mobile Mouse Lite
7. Remote Control Collection

We installed each of the above clients on an Android phone and the corresponding server on a remote machine running Windows, OS X, or Linux. We used two types of tools for reverse engineering the messaging protocols:

1. Wireshark - Wireshark is a network protocol analyzer. Using Wireshark, we were able to analyze incoming and outgoing packets from the remote machine. This was extremely useful for applications using plaintext protocols.
2. Decompilers - Tools like Java Decompiler Project, JetBrains dotPeek, Decompile Android, and dex2jar allowed us to decompile many of the client APK files and server executables into very readable Java code.

After reverse engineering the protocols, we set up virtual machines running the above applications and send arbitrary keyboard and mouse commands from the host machine via UDP and TCP. For example, one of our exploits can open a web browser and visit arbitrary websites, while another can download and execute files.

In addition, we also created nmap probes for five of the vulnerable servers. The probes allow a user to identify if a host is running one of these applications through a port scan. Only one of the exploitable servers never responded with unique, fingerprintable messages, making a probe infeasible. The probes have been submitted to the nmap project, although no feedback about their inclusion in future versions has been received.

Application-Specific Analysis

Teamviewer

Of the clients we tested, TeamViewer is by far the most robust. It has 10 - 50 million installs on the Google Play app market, and as one would hope, uses standard public-key encryption and AES session encoding. We tried without success to compromise machines running TeamViewer servers.

Air HID

Air HID's server is written in Java and had no obfuscation, so it was easily decompiled into readable source code using the Java Decompiler tools. It uses an unencrypted UDP-based protocol with no support for authentication, meaning that any machines that use Air HID are vulnerable with minimal effort.

Analyzing the Air HID protocol shows that the server has first-class support for ping commands (which conveniently return the server version), mouse movement and clicks, key presses, opening arbitrary URLs and setting clipboard data. We were able to write a Python-based driver that exploits most of this functionality to take over a machine running Air HID.

Remote Mouse

Reverse-engineering Remote Mouse's client and server was more difficult than most other apps due to the fact that the app is written in obfuscated Java, and the server is written in obfuscated C#. However, since numbers and strings are difficult to obfuscate, they were left unchanged, so we were able to deduce what some of the code does by reading logging messages, error messages and looking for known integer constants such as the port number. We also found that the Android app does not in fact handle the Remote Mouse protocol -- instead, it calls into a DLL written in unobfuscated C# which we were able to decompile into readable source code.

On the surface, the application offers authentication and some appearance of security. However, its homespun crypto schemes are easily broken, and could even enable attacks outside the application by leaking data about the user's password.

In its authentication, the server issues a challenge-response session with the client. The server sends the client a string, and the client must reverse the string, and hash it with MD5 with no salt as the response. No secret is required, which makes this scheme trivially breakable. The string the server sends is also always constant -- the unsalted MD5 of the user's password -- so an

attacker can then use rainbow tables to figure out what the exact password is.
We also found that Remote Mouse partially encrypts keystroke data -- when it sends data about printable characters, it sends the ASCII code of the character XORed with the constant byte value 53. This was a prime example of security by obscurity, and we were easily able to discover and circumvent this layer of protection by decompiling the server. Furthermore, control sequences such as Enter and the function keys were sent in plain text even though the source code also contains a function capable of “encrypting” them in a similar fashion.

We also found a component of the server that seems overtly malicious -- Remote Mouse secretly opens up an HTTP server running on port 1985 which serves the host’s My Pictures folder (or equivalent on Mac OSX). This behavior is not documented in the application and is not used by the client. As there is no authentication required to access the HTTP server, this means that any files in the host’s My Pictures folder are publicly shared. Remote Mouse also includes a feature that generates a QR code with your server’s IP address for easy connections, but this QR code is generated by sending a GET request to a page located on the application’s website: http://www.remotemouse.net/qrcode.php?ip=<ip address>. These two features put together allow the application developers to determine whether a Remote Mouse server is behind a NAT (by comparing the originating address of the request with the IP address parameter) and if not, use the HTTP server to retrieve all the files in the host’s My Pictures folder. We were unable to ascertain for sure whether the application developers had any malicious intent, but this combination of features is definitely suspect at best.

WiFi Mouse

WiFi Mouse is a relatively simple software solution that has minimal customization options. It does not offer password or encryption schemes for the user, and therefore is easily spoofed. Furthermore, its C++ server source is posted on their download website, so its protocol is clearly documented. We were therefore able to establish full control over machines running the server with minimal effort.

Additionally, we found that the WiFi Mouse developers included malware in their Windows installer. This is particularly concerning, considering that this application is one of the more popular ones we examined, with an install base of 1-5 million users in the Play store alone.

Android Mouse and Keyboard

We used JD-GUI to decompile the server JAR file. The client sends over mouse and keyboard commands in plaintext via UDP to a specified server-side port. For example, a packet reading “2000x-2000yMVEX” would instruct the server to move the mouse by 2000 pixels in the x direction and -2000 pixels in the y direction.

In addition, there is a password feature for this application that doesn’t seem to provide any security. It appears that the password checking is done client-side, so it is bypassed simply by sending UDP commands to the server as above. The password (in plaintext) can be retrieved from the server by establishing a TCP connection to the same port and sending

“AMPASSCHECK\n”. The server responds with the password, which is checked client-side against the user-entered password. In an entertaining way, the password only proves to slow down the actual user, but an adversary can simply ignore it. We were therefore able to establish full control over machines running this application.

Mobile Mouse Lite

We found that Mobile Mouse Lite’s client sends over mouse and keyboard commands in plaintext via TCP to a specified server-side port. We were able to write a driver that can send these commands to take control of a host running this application.

In addition, if the server requests a password, the client sends its password attempt in plaintext, which is susceptible to a man-in-the-middle attack. Upon establishing a connection with the server, the server responds with the name of the server OS, the name of the machine, the MAC address, and the OS version number.

Remote Control Collection

We were unable to write a full driver for Remote Control Collection, mostly due to its bizarre network protocol. Security through obscurity may not be a sufficient form of protection, but RCC may have perfected it. For example, it establishes two ports on with the server, one TCP, one UDP. A “left mouse click” action is triggered by sending a 5 byte code associated with “left mouse down” over TCP, and then sending another 5 byte code associated with “left mouse up” over UDP. This behavior, as well as its touchscreen-dependent mouse movement protocol, proved to be very difficult to emulate, and while the driver is capable of some actions, is not as reliable as the others.

RCC was the only application that is not fingerprintable because its server never responds with messages that contain unique text. The only data originating from the server were ACKs and packets sent to establish the TCP connection.

The Code

For each of the six compromised remote desktop applications, we created a driver in Python, allowing an adversary to specify the victim machine’s IP and port number to send arbitrary keyboard and mouse commands. We wrote proof of concept scripts that could be sent over a driver to a remote machine, that demonstrate the potential impact of arbitrary mouse / keyboard input.

Our code for these drivers can be found here: https://github.com/obi1kenobi/pyre

Conclusion

We found that the most popular mobile remote control applications have deeply flawed security schemes. This is concerning considering that they provide direct access to their host machines, and casual users might install them without second thought. The vulnerable applications we

discovered have between 2-7.5 million downloads in the Play Store alone (several have clients of iOS as well, but the Apple Store doesn't provide download numbers).

To highlight this danger, we created drivers for six of the most popular remote control applications in the Play Store, and proof of concept exploits that could be run on any of them to take over a machine. We wrote nmap fingerprints so that these applications could be identified from a port scan. An attack scenario might be using nmap to scan a large range of IPs, and when a vulnerable host is found, an appropriate app driver is selected, and an exploit run on the machine. Because this process can be entirely automated, this could be applied to the entire IPv4 address space.

If you want to remotely control your computer but not allow adversaries to do the same, you should use established cryptographic protocols to properly authenticate clients and encrypt all traffic; however, of all the apps we looked at, only the TeamViewer application followed these rules of thumb. We also show that code obfuscation is not sufficient protection against protocol reverse-engineering, as it is difficult to do correctly and only slightly lengthens the process of reverse engineering.