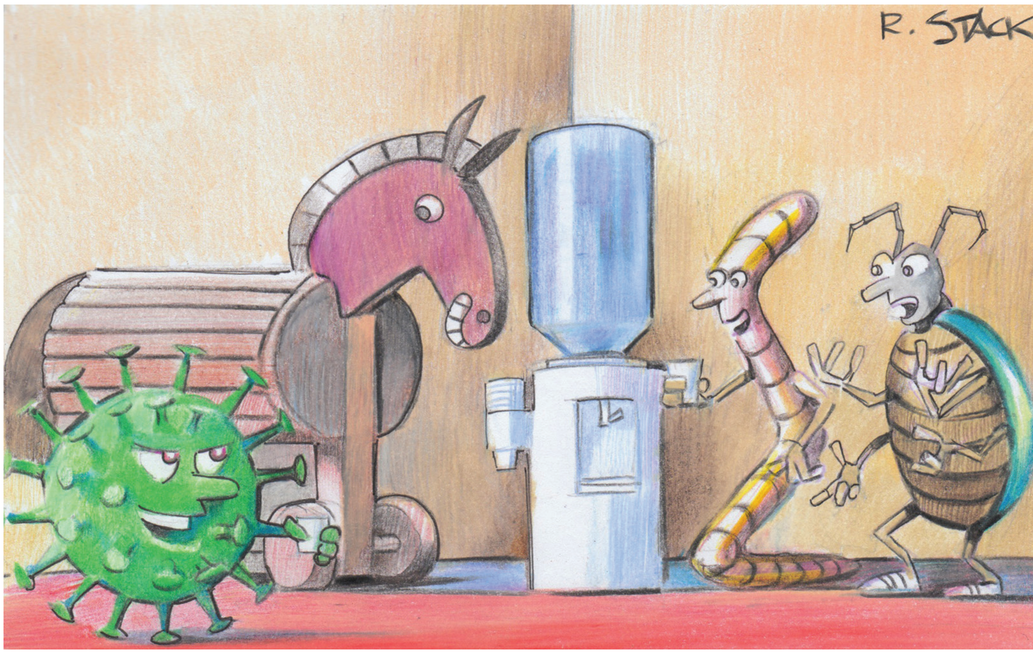
Editor: Jonathan Margulies, jonathan@qmulos.com

Colluding Apps:

Tomorrow's Mobile Malware Threat

Atif M. Memon | University of Maryland, College Park
Ali Anwar | Montgomery Blair High School



Mobile devices (tablets, smartphones, watches, and wearable gadgets) carry a wealth of personal and professional data that software apps can read, access, and modify. Unfortunately, some apps are malicious, stealing users' data and transmitting it without their knowledge. Given that an estimated 7.22 billion mobile devices are in use—a number rivaling the human population[1]—and that mobile platforms are increasingly reporting malware,[2,3] mobile malware might put our privacy at unprecedented risk.

App Isolation and Interaction

Today's mobile OSs are designed with a focus on security. Android, the most popular mobile OS, isolates each app in an *application sandbox* by leveraging the security features of its underlying Linux kernel. Each app runs in its own memory space, has access to a permission-protected file system, and has protected CPU cycles. Unless the user explicitly bypasses it, this sandbox design protects apps from interfering or interacting with one another and other vital system components. For example, a banking app can't access files from a messaging app and vice versa.

Although it successfully isolates apps from one another, the sandbox design doesn't completely preclude malware exploitation. For example, an app might intentionally or unintentionally leak the device's GPS coordinates by encoding them in the URL of an HTTP request. Android provides additional mechanisms to protect against such leaks. Each protected feature of the device (such as GPS or the network) requires explicit access permission. Hence, for an app to leak GPS location information over the network, it must have simultaneous permission to ACCESS_FINE_LOCATION for GPS and access to INTERNET for the network. Only users can grant these permissions when they install the app. However, because most users ignore permission warnings at app install time,[4] they might end up installing overprivileged apps—those requesting more permissions than they need to do their job[5]—some of which might be malicious.

It's not practical to completely isolate apps from one another and the system. Numerous use cases require that apps be allowed (and even encouraged) to interact; for example, a messaging app might need to interact with the device contacts app and the camera app so users can capture and send pictures to their contacts. Moreover, apps might invoke parts of other apps to enhance the user experience. For example, an app that wants to show a map as part of its user interface doesn't need to write map-viewing code from scratch. Instead, it can

Covert Communication Channels

Consider two apps: *FFitt* and *IIMsgg*. FFitt collects and maintains fitness data locally on the user's mobile device. It has permission to communicate unshared local storage, device location, and system settings via Bluetooth to a fitness device. It has no Internet access. IIMsgg sends and receives text messages via an Internet-based messaging service. It has permission to access the Internet, contacts, and system settings. Security analysis tools would deem both apps safe because there's no way to leak sensitive fitness and location data.

Unbeknownst to these analysis tools is that there's malicious code (a snippet is shown in Figure A) embedded in the apps by developers or a hacked integrated development environment[1] that allows them to communicate via an unconventional channel: the device's screen-off time-out. In Android, this is the time in milliseconds before the device goes to sleep or begins to dream after a period of inactivity. Apps are allowed to read and modify this value. Using this mechanism, the FFitt app covertly sends fitness and location data to the IIMsgg app, which then leaks this information to a contact via a message. Figure A implements an oversimplified but illustrative protocol that allows FFitt to transmit its sensitive data as a set of numbers (`message` in the code). FFitt encodes each number as a time-out value and sets it using `Settings.System.putInt` (full code not shown due to space); IIMsgg reads the value (`Settings.System.getInt`) and resets it to indicate successful receipt. FFitt then "sends" the next number. This process continues until all the numbers have been transmitted. This example illustrates how two apps' shared resource (screen time-out) might be used as a covert communication channel to effectively complete a path from a source of sensitive data (location, fitness data) to an external sink. None of today's malware tools would detect this collusion.[2]

References

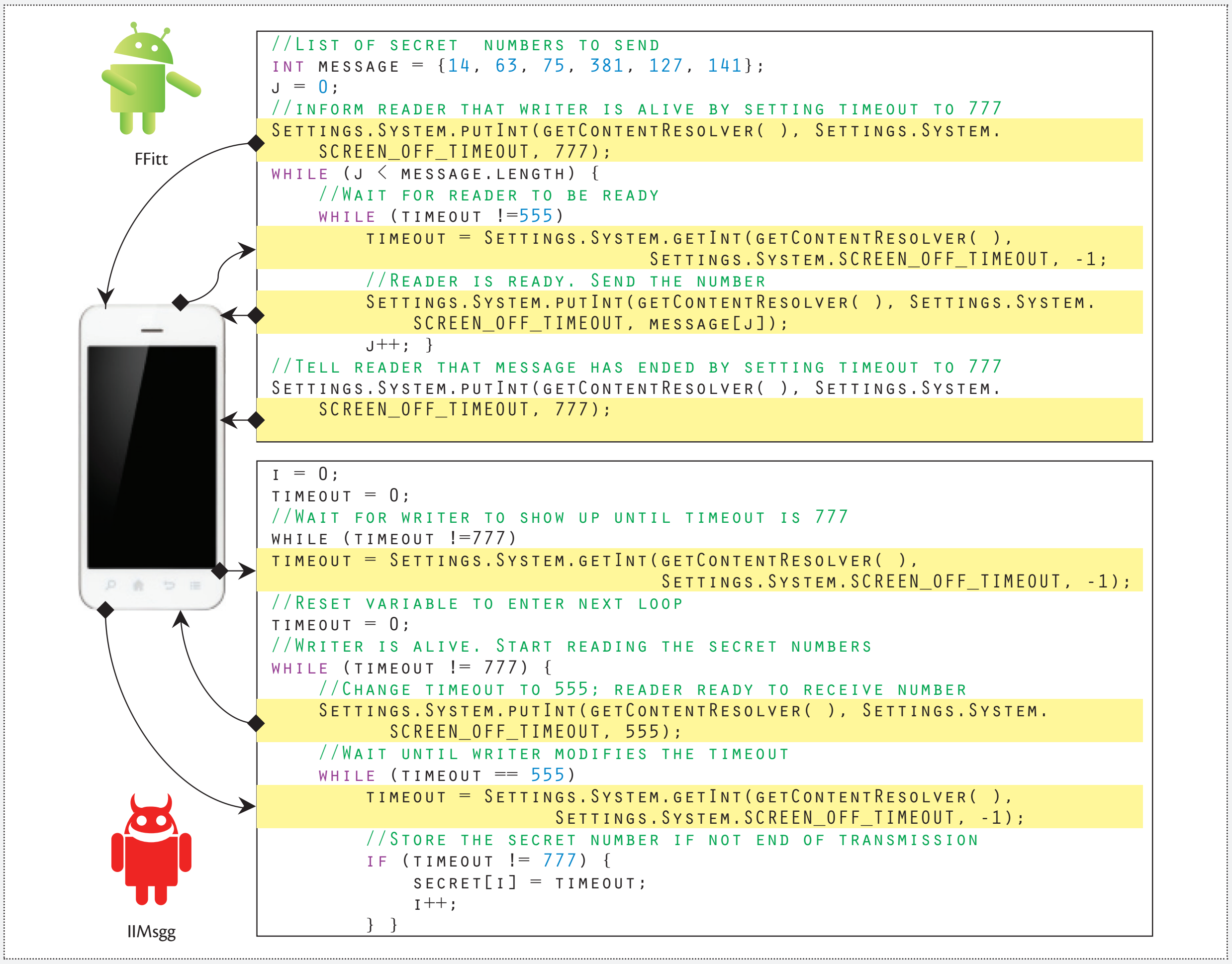
1. "Novel Malware XCodeGhost Modifies XCode, Infects Apple iOS Apps and Hits App Store," Palo Alto Networks, 17 Sept. 2015; http://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store.
2. D.J.J. Sufatrio et al., "Securing Android: A Survey, Taxonomy, and Challenges," *ACM Computing Surveys*, vol. 47, no. 4, 2015, pp. 58:1–58:45.

invoke the default map app's map screen (similar to a subroutine); users can then view the map, close it, and return control back to the calling app. In another example, users might want a certain app to be invoked for a system event such as a text's arrival.

To cater to such use cases, Android allows apps to interact with one another and the system, pass data, return results, and share resources; apps need to explicitly allow for such interaction via permissions. However, allowing interaction such as communication via messages (implemented as Intent objects in Android) opens the door to malware. For example, the DroidDreamLight malware used receipt of the android.intent.action.PHONE_STATE Intent as its trigger—such as when users receive a phone call. Once triggered, this malware executes its own code.[6]

Detecting and Removing Mobile Malware

The Internet abounds with advice and best practices for avoiding mobile malware. Yet malware continues to proliferate, indicating that prevention alone isn't sufficient. Several techniques can detect and remove mobile malware. First, *basic static techniques* check the app's attributes such as file name, checksums or hashes, file type, and file sizes. Any discrepancies are flagged as potential malware. Second, *static code search techniques*, such as those used in virus scanners, search for syntactic signatures in the app's code. Using a database of code sequence patterns that are



```
//List of secret  numbers to send
int message = {14, 63, 75, 381, 127, 141};
j = 0;
//inform reader that writer is alive by setting timeout to 777
Settings.System.putInt(getContentResolver( ), Settings.System.
    SCREEN_OFF_TIMEOUT, 777);
while (j < message.length) {
    //Wait for reader to be ready
    while (timeout !=555)
        timeout = Settings.System.getInt(getContentResolver( ),
                                    Settings.System.SCREEN_OFF_TIMEOUT, -1;
        //Reader is ready. Send the number
        Settings.System.putInt(getContentResolver( ), Settings.System.
            SCREEN_OFF_TIMEOUT, message[j]);
        j++; }
//Tell reader that message has ended by setting timeout to 777
Settings.System.putInt(getContentResolver( ), Settings.System.
    SCREEN_OFF_TIMEOUT, 777);
```

```
i = 0;
timeout = 0;
//Wait for writer to show up until timeout is 777
while (timeout !=777)
timeout = Settings.System.getInt(getContentResolver( ),
                                Settings.System.SCREEN_OFF_TIMEOUT, -1);
//Reset variable to enter next loop
timeout = 0;
//Writer is alive. Start reading the secret numbers
while (timeout != 777) {
    //Change timeout to 555; reader ready to receive number
    Settings.System.putInt(getContentResolver( ), Settings.System.
          SCREEN_OFF_TIMEOUT, 555);
    //Wait until writer modifies the timeout
    while (timeout == 555)
        timeout = Settings.System.getInt(getContentResolver( ),
                        Settings.System.SCREEN_OFF_TIMEOUT, -1);
        //Store the secret number if not end of transmission
        if (timeout != 777) {
            secret[i] = timeout;
            i++;
        } }
```

Figure A. A snippet of malicious code embedded in the FFitt and IIMsgg apps that lets them communicate via an unconventional channel: the device's screen-off time-out. FFitt encodes sensitive fitness and location data as time-out values, which IIMsgg then sends to a contact via a message.

considered malicious, the technique identifies a program as malware if part of its code matches a pattern in the database. Third, *static code analysis techniques* perform flow analysis of the app's code to check whether there's a control flow path that allows the app to access and leak sensitive information to an external entity.

These static techniques are severely limited because today's mobile apps increasingly use dynamic runtime mechanisms such as virtual function calls, dynamic class loading, reflection, multithreading, and event handler callbacks. Such mechanisms necessitate a fourth technique: *dynamic analysis*. This technique manually or automatically runs the app and observes its runtime behavior; for example, it monitors system processes, file-system and registry changes, and network activity. Dynamic analysis is typically done in a safe (most likely emulated) environment in which the analyst can run the malware and observe its behavior without interference from other apps. However, this technique is incomplete because it's generally impossible to run a software program on all its possible inputs. Moreover, because dynamic analysis involves actually executing the app, the app might fool the analysis by "turning off" malicious behavior in certain configurations (such as for particular platforms, locations, time, and devices). For example, the Dendroid malware used emulation-detection code to successfully evade Google Bouncer, an automated app-vetting tool.[7]

App Collusion

Even as the security community is starting to understand and detect individual malicious apps, a new threat is emerging: *colluding apps*.[8] In collusion attacks, a malicious operation is broken into smaller parts and distributed across multiple apps. These apps communicate (or wait for a signal) to play their small, individually undetectable roles in the operation. Each app avoids suspicion by requesting the minimum permissions needed for its role. For example, when two apps collude, the first app might read sensitive data and transmit it to the second app, which transmits it to the outside world. Analyzed individually, the apps would be considered benign because there's no direct path from sensitive data to its transmission. In an effort to detect colluding apps, recent research has extended flow analysis to include app message-passing channels.[9] These messages can be monitored at runtime to identify app pairs that exchange messages and, hence, possibly collude.[10]

In the "Covert Communication Channels" sidebar, the example of collusion between the FFitt and IIMsgg apps demonstrates the serious and complex nature of collusion. In 1973, Butler Lampson identified a general form of this "confinement problem,"[11] although its application to today's mobile apps gives it new light. Malware detection tools considering each app in isolation have no hope of detecting collusion because they aren't designed to analyze sets of apps simultaneously. Even if new tools could target collusion, they'd be limited in at least two ways. First, they wouldn't know which apps to analyze together: there are millions of apps in the marketplace, and any two (or more) might be malicious and colluding. To analyze all possible app pairs, the tool would need to analyze N^2 pairs, where N is the number of apps in the marketplace. Analyzing all possible triples—to detect sets of three colluding apps—would require N^3 runs. Thus, the cost of analysis grows exponentially with the number of concurrently analyzed apps. Second, the analysis wouldn't know which communication channels to intercept.

Apps share dozens of resources, each with multiple attribute values that apps are allowed to read or modify (including network status, sound volume level, device orientation, Bluetooth status, USB connection, and altitude). Any of these resources might be used—individually or together—as covert communication channels. In principle, a successful analysis tool must monitor all possible communication channels, which means monitoring every possible code path to every read/write of a shared resource's attribute. This is a practically impossible task.

Potential Security Improvements

The problem of colluding malware hits at the very core of today's mobile OS security model: individually restricting apps (for example, via permissions or sandboxing) is sufficient for their safe composition on a single device. This model is inadequate in light of colluding malware. Thus, we must revise and enhance the model, which will involve an expensive, major rewrite of the security components of today's mobile OSs. Detecting colluding apps remains an open research problem, the solution to which eludes both practitioners and researchers. We believe that any solution must address two fundamental challenges: which covert channels to examine and which sets of apps to analyze together for collusion.

To identify covert channels, we propose the following steps:

- Exhaustively list all possible shared resources and their attributes that apps can access and modify (originally called "shared-resource matrix methodology"[12]).
- Examine the code of all apps in today's mobile marketplace to determine the shared resources and attributes being accessed in practice.

To identify which sets of apps to analyze together for collusion, we propose the following approaches:

- Examine app advertisements that ask users to download other apps.
- Mine social media postings and Internet sites that ask users to download apps. For example, 3 million Minecraft fans—looking for cheat sheets and playing tips—were tricked into downloading more than 30 fake apps.[13]

There's no way to tell how many covert-channel colluding apps are in the mobile marketplace, already stealing our information. Many users are unaware that their devices have even been compromised. Indeed, if rogue nations use malware to spy on other countries' government agents, businesses, and diplomats to gain strategic advantage, then the results of exploits might not translate to a traceable outcome (such as a credit card charge)—meaning that the malware can go undetected for years.

Given that shared resource attributes are easily repurposed as covert communication channels, we believe that we're on the edge of a massive influx of apps using such channels to go undetected for long periods. It's not difficult to imagine the developers of today's individual malicious apps splitting their malware code across multiple and seemingly benign apps, using a covert channel to communicate between the apps, and successfully performing malicious operations. If their exploits are discovered, they can quickly move the malware to other apps, change the covert communication to an alternative shared resource, and repenetrate the marketplace. This practice could continue indefinitely, until we develop better solutions to the problem of malware avoidance and detection. ■

References

1. "There Are Officially More Mobile Devices than People in the World," *Independent*, 7 Oct. 2014; www.independent.co.uk/life-style/gadgets-and-tech/news/there-are-officially-more-mobile-devices-than-people-in-the-world-9780518.html.
2. "Protect Your Android Device from Malware," CNET, 25 June 2014; www.cnet.com/how-to/protect-your-android-device-from-malware.
3. "McAfee Labs Threats Report May 2015," McAfee, May 2015; www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf.
4. A. Porter Felt et al., "Android Permissions: User Attention, Comprehension, and Behavior," *Proc. 8th Symp. Usable Privacy and Security* (SOUPS 12), 2012, pp. 3:1–3:14.
5. X. Wei et al., "Permission Evolution in the Android Ecosystem," *Proc. 28th Ann. Computer Security Applications Conf.* (ACSAC 12), 2012, pp. 31–40.
6. M. Balanza et al., "DroidDreamLight Lurks behind Legitimate Android Apps," *Proc. 6th Int'l Conf. Malicious and Unwanted Software* (MALWARE 11), 2011, pp. 73–78.
7. "Dendroid Spying RAT Malware Found on Google Play," Help Net Security, 3 July 2014; www.net-security.org/malware_news.php?id=2726.
8. C. Marforio et al., "Analysis of the Communication between Colluding Applications on Modern Smartphones," *Proc. 28th Annual Computer Security Applications Conf.* (ACSAC 12), 2012, pp. 51–60.
9. D. Sbîrlea et al., "Automatic Detection of Inter-application Permission Leaks in Android Applications," *IBM J. Research and Development*, vol. 57, no. 6, 2013, pp. 2:10– 2:10.
10. K.O. Elish, D. Yao, and B.G, Ryder, "On the Need of Precise Inter-app ICC Classification for Detecting Android Malware Collusions," *Proc. IEEE Mobile Security Technologies* (MoST)/*IEEE Symp. Security and Privacy* (SP), 2015.
11. B.W. Lampson, "A Note on the Confinement Problem," *Comm. ACM*, vol. 16, no. 10, 1973, pp. 613–615.
12. R.A. Kemmerer, "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels," *ACM Trans. Computer Systems*, vol. 1, no. 3, 1983, pp. 256–277.
13. "Minecraft Cheats Scareware Apps Affect 600,000 Users," WCCF Tech, May 2015; http://wccftech.com/minecraft-cheats-scareware-apps-affect-600000-users.

Atif M. Memon is a professor in the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland, College Park. Contact him at atif@cs.umd.edu.

Ali Anwar is a student at Montgomery Blair High School. Contact him at alia7477@gmail.com.