FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment

Earlence Fernandes, Bruno Crispo, *Senior Member, IEEE*, and Mauro Conti, *Member, IEEE*

***Abstract*—Many modern smartphones and car radios are shipped with embedded FM radio receiver chips. The number of devices with similar chips could grow very significantly if the U.S. Congress decides to make their inclusion mandatory in any portable device as suggested by organizations such as the RIAA. While the main goal of embedding these chips is to provide access to traditional FM radio stations, a side effect is the availability of a data channel, the FM Radio Data System (RDS), which connects all these devices. Different from other existing IP-based data channels among portable devices, this new one is open, broadcast in nature, and so far completely ignored by security providers. This paper illustrates for the first time how to exploit the FM RDS protocol as an attack vector to deploy malware that, when executed, gains full control of the victim's device. We show how this attack vector allows the adversary to deploy malware on different platforms. Furthermore, we have shown the infection is undetected on devices running the Android OS, since malware detection solutions are limited in their ability due to some features of the Android security model. We support our claims by implementing an attack using RDS on different devices available on the market (smartphones, car radios, and tablets) running three different versions of Android OS. We also provide suggestions on how to limit the threat posed by this new attack vector and explain what are the design choices that make Android vulnerable. However, there are no straightforward solutions. Therefore, we also wish to draw the attention of the security community towards these attacks and initiate more research into countermeasures.**

***Index Terms*—Android security, smartphone security, FM radio, novel attack vectors.**

I. Introduction

MINIATURIZATION and economy of scale is making it possible to embed computational power, communication capabilities and storage capacity in many new products (i.e., domestic appliances, cars, apparel, etc.). Users can be online and interact virtually at any time and from anywhere, by means of devices such as smartphones, tablets, eReaders, etc. Many of these devices rely on the Internet Protocol (IP) for data communications. Thus, the sender needs to know the recipients' address to be able to reach it.

For devices that do not support IP, like feature phones, alternative methods like the Short Message Service (SMS) can be used to send data. However, SMSs also require knowledge of the recipient's address—the mobile phone number.

Most recently though, device manufacturers started embedding FM Radio receiver chips in devices such as smartphones and car radios. The side effect of this decision is the availability of a new data communication channel, the FM Radio Data System, that different from existing ones is open and broadcast so senders do not need to know the recipients' address to be able to communicate with them. Furthermore the system can be used to reach all devices located in a specific geographical area. While this decision affects millions of devices, the impact can be even much wider as the U.S. Congress is currently addressing the issue of mandating the inclusion of FM receivers in all portable electronics [18]. Thus affecting many other devices such as alarm clocks, digital photo frames, music players, etc.

This paper shows for the first time, how the FM Radio broadcast data channel can be exploited as a new and effective attack vector to deploy malware. Different from other IP-based attack vectors, this one does not require port scanning. Furthermore, once the malware is downloaded on the victim's device, the attack leaves very few explicit information to trace back the source of the attack. Moreover, the attack can be geographically targeted and does not require network access. Also, it can be executed at a time and place of the attacker's choosing. By means of the FM Radio broadcast channel, malware can potentially infect a larger number and a wider range of devices compared to existing malware deployment techniques.

We have implemented concrete examples of attacks on different devices: smartphones, car radios and tablets all running (different) versions of the Android OS.

We demonstrate how to split the malware into two pieces to evade malware detection. A victim is first tricked into installing a benign looking app that uses the RDS interface; this app contains no malicious behavior at installation time. Thus, even a strict vetting process would not find any malicious behavior in the code of the app. The attacker can easily publish the application on the Android Market. Then, the app dynamically downloads the back door that allows it to reassemble RDS packets and execute the payload. Dynamic download of updates and patches is a common practice in the Android world done by means of a

Manuscript received July 25, 2012; revised December 20, 2012; accepted April 09, 2013. Date of publication April 24, 2013; date of current version May 16, 2013. The work of M. Conti was supported by a Marie Curie Fellowship funded by the European Commission for the PRISM-CODE project (Privacy and Security for Mobile Cooperative Devices) under the agreement PCIG11-GA-2012-321980. This work was supported in part by the TENACE PRIN Project (20103P34XC) funded by the Italian Ministry of Education, University and Research. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Miodrag Potkonjak.

E. Fernandes was with Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. He is now with the University of Michigan, Ann Arbor, MI 48109 USA (e-mail: earlence@umich.edu).

B. Crispo is with Università di Trento, Trento, 38122, Italy (e-mail: crispo@disi.unitn.it).

M. Conti is with the University of Padua, Padua, 35131, Italy (e-mail: conti@math.unipd.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2013.2259818

well known API, the `DexClassLoader` [5] that we used as well. We also show how to distribute an existing exploit payload over the RDS interface such that when executed gets root access to the infected device.

To be able to mount this attack we reverse engineered the FM Radio API of the target devices.

As we explain in Section VI, the exploit is undetected due to a feature in the Android security model Antiviruses cannot scan what an application downloads at runtime. As a matter of fact, the devices we attacked, with the exception of the car radio, for which we did not find any antivirus, were running antivirus solutions (free and commercial) and no alarm was raised.

We conducted these attacks for scientific purposes and with a limited budget, however similar techniques and vulnerabilities can be exploited in a much more effective and dangerous way by organized crime or attackers sponsored by governmental agencies. Moreover, these attacks have highlighted several important and open security issues.

a) Contributions: The main contributions of the paper are the following:

- We show for the first time, how the FM Radio broadcast data channel can be exploited as a new and effective attack vector to deploy malware.
- We implemented concrete examples of attacks for different HW platforms: smartphones, car radios and tablets, running different version of the Android OS.
- We reverse engineered the FM Radio API of the target devices, highlighting a security problem with the implementation of such API, which allows the attacker to bypass the native Android OS security checks.
- We provide an analysis on the current limitation of the effectiveness of Antivirus software running on Android OS in detecting malware download at runtime.
- Finally, we suggest possible countermeasures that could help to prevent or at least limit the effect of the presented attacks.

b) Organization: The rest of the paper starts with background information on RDS and Android security (Section II). Following that, FM RDS-Based attacks are introduced along with a description of the Threat Model (Section III). Then, we detail real attacks on different devices (Section IV), along with technical information on how they were executed. We follow these with possible optimizations and countermeasures (Sections V and VI). We then review the most closely related research (Section VII) and we conclude in Section VIII.

II. Background

This section contains some preliminary concepts and background knowledge used in the rest of the paper.

A. Radio Data System (RDS)

FM (Frequency Modulation) broadcasts are a very popular and widespread form of entertainment. The FM broadcast standard includes the Radio Data System (RDS) protocol. The corresponding U.S. version is Radio Data Broadcast Standard (RDBS). Radio stations use these widely adopted protocols to transmit small amounts of data to receivers. These data include program name, alternative frequencies and traffic congestion updates. Additionally, RDS has been adapted to be used in emergency warning systems [1]. Recent research has proposed a mechanism to provide value added services over RDS [36].

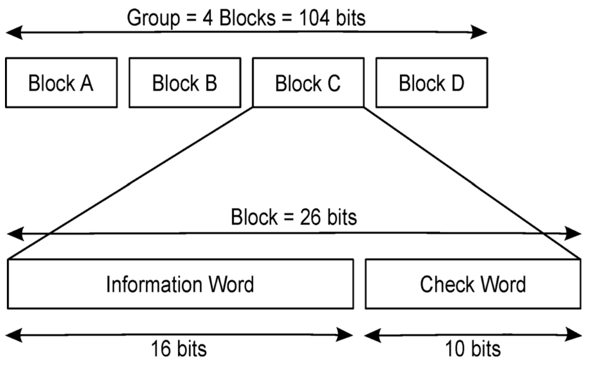

Fig. 1. Baseband coding.

RDS carries information on a 57 KHz signal. This is technically referred to as a subcarrier. The digital data sent over RDS is organized as groups. As shown in Fig. 1, we can see that each group is 104 bits in length. Each group is divided into 4 blocks (in Fig. 1, these are blocks A, B, C and D). There are several types of groups, based on the organization of bits in a group. Type 2A/2B is known as *RadioText* (RT) and carries arbitrary ASCII text. Group 0A/0B is known as *Program Service* (PS). PS name refers to the name of the current radio station. All groups will contain a Program Identification value (PI) which is used to identify the country where the transmission takes place. The data rate of RDS is 1187.5 bits/s. It takes 87.6 ms to transmit one group. RDS has an inbuilt error correction mechanism which can detect single and double bit errors in a block. Additionally, it can detect a single error burst spanning 10 bits or less.

B. Android Overview

Google Android is an open source Linux-based mobile platform developed by the Open Handset Alliance (OHA). We chose Android as target of our analysis since it is the fastest growing OS. Android runs on a long list of different devices such as smartphones, tablets, TVs, car entertainment systems, and in-flight entertainment systems of the Boeing 787 Dreamliner [17], just to name a few. Most of the Android applications are programmed in Java and compiled into a custom byte-code that is run by the Dalvik Virtual Machine (DVM). In particular, each Android application is executed in its own address space and in a separate DVM. Android applications are built combining any of the following four basic components. *Activities* represent a user interface; *Services* execute background processes; *Broadcast Receivers* are mailboxes for communications within components of the same application, or belonging to different applications; *Content Providers* store and share application's data. Application components communicate through messages called *Intents*.

C. Android Security Model

Focusing on security, Android combines two levels of enforcement [27], [40]: at the Linux system level and the application framework level. At the Linux system level Android is a multiprocess system. During installation, an application is assigned a unique Linux user identifier (`UID`). Additionally, an application can be made a member of a group to enable the application to access certain resources like the network interface. Thus, in the {Android OS} each application is executed as a different user process within its own, isolated, address space and storage space. Moreover, these applications execute in user space and there is no way in which they can elevate their privileges. These mechanisms constitute the Android sandbox.

At the application framework level, Android provides access control through the Intercomponent Communication (ICC) reference monitor. The reference monitor provides Mandatory Access Control (MAC) enforcement on how applications access the components. In the simplest form, protected features are assigned with unique security labels—*permissions*. Protected features may include protected application components and system services (e.g., Bluetooth). To make use of protected features, the developer of an application must declare the required permissions in its package manifest file—`AndroidManifest.xml`.

As an example, consider an application that needs to monitor incoming SMSs, `AndroidManifest.xml` included in the application's package would specify: $\langle$ `uses` $-$ `permission android` $:$ `name` $=$ ``` `` ```android.permission.RECEIVE_SMS"/$\rangle$. Permissions declared in the package manifest are granted at the installation time and cannot be modified later. Each permission definition specifies a protection level which can be: `normal` (automatically granted), `dangerous` (requires user confirmation), `signature` (requesting application must be signed with the same key as the application declaring the permission), or `signature or system` (granted to packages signed with the system key).

III. FM RDS-Based Attacks

Here we present an overview of our FM RDS-Based attacks and how we implemented them. We describe the threat model we considered. Furthermore, we provide a description of the software and equipment required to mount such an attack.

A. Threat Model

Our attacks assume the adversary owns equipment that is capable of broadcasting custom RDS data. To send the custom data to receivers, all that is required is an FM transmitter that can overpower the transmission on a frequency which is used by a real station, which is authorized to transmit on that frequency. Important, the attacker does not require a IP-based communication channel to talk to the device. and the victims' device are not required to have network enabled.

We bought for few hundred U.S. dollars, a transmitter that is highly mobile, easily fits into a backpack, powered by a 11 Volt battery and can cover a radius of 3.5 miles. This area is sufficiently large, so as to attack a large number of devices. Our transmitter complies with existing regulations and laws. We should not expect the same behavior from a real attacker. Thus, the transmission range can be higher to cover a wider area than we were able to do.

Its worth mentioning that an attacker can intrude on legal broadcasts from a radio station (in a way similar to TV signal intrusion reported in [16]) and broadcast custom RDS data to all devices listening to that radio station. This has the potential to attack a larger number of devices, as compared to our prototype.

Another assumption is that the attacker publishes an application on the Android market (or other suitable software distribution point specific to the desired target). This application, besides performing its functions, will also execute exploits we transmit over RDS. The API used by our application to execute the exploit is normally used by many other legitimate application to download updates and extensions at runtime (e.g., player skins) after their installation. So an automated code review or a behavioral analysis will not be able to flag our application as malicious.

As an example of distribution sources, consider third party marketplaces that provide Android applications. Recent work [42] has highlighted the issue that many of these marketplaces have repackaged applications and hence, it is not easy for a user to distinguish between the real and repackaged applications. To increase the download count of his application, the attacker can choose to repackage a well-known application and masquerade it on these marketplaces.

B. Attack Overview

In this section, we provide a general description of how we implemented our attack. As we have seen in Section II-A, *RadioText* carries arbitrary textual data. We leverage the *RadioText* field to transmit an exploit that will enable us to gain control of the target by a privilege escalation attack. The attack is mounted in three phases.

Phase 1: The attacker has two main tasks which represent the two paths through which we mount the attack. The first is to create an apparently honest app (Step 1a, Fig. 2) and upload it to a popular distribution point. The second task is to obtain a privilege escalation exploit for the desired targets (Step 1b). Since the bandwidth of the RDS protocol is very low, we need to packetize the exploit, and attach sequence numbers to them (Step 1c). This packetization step basically breaks up a multikilobyte binary payload (more detail in Section IV-C) into several smaller Base64 encoded packets. The packets are modulated as FM and broadcast via an FM Transmitter (Step 1d). Whether an audio track is transmitted does not matter, as the receivers can receive *RadioText* without the need to play the audio. In case the user is listening to a track, we can easily transmit a popular audio track, which is identical to the one being played on a real FM channel.

There are a few options to choose from to select a transmission frequency. The most obvious is to scan the spectrum for an unused frequency and start transmitting on that. In our prototype attack, we do this. The other is to hijack an alternative frequency of an existing channel, as done in [19]. Basically, it relies on the FM Capture Effect, wherein only one of the stronger of two signals at, or near the same frequency will be demodulated. Another option is Broadcast Signal Intrusion. This is a common

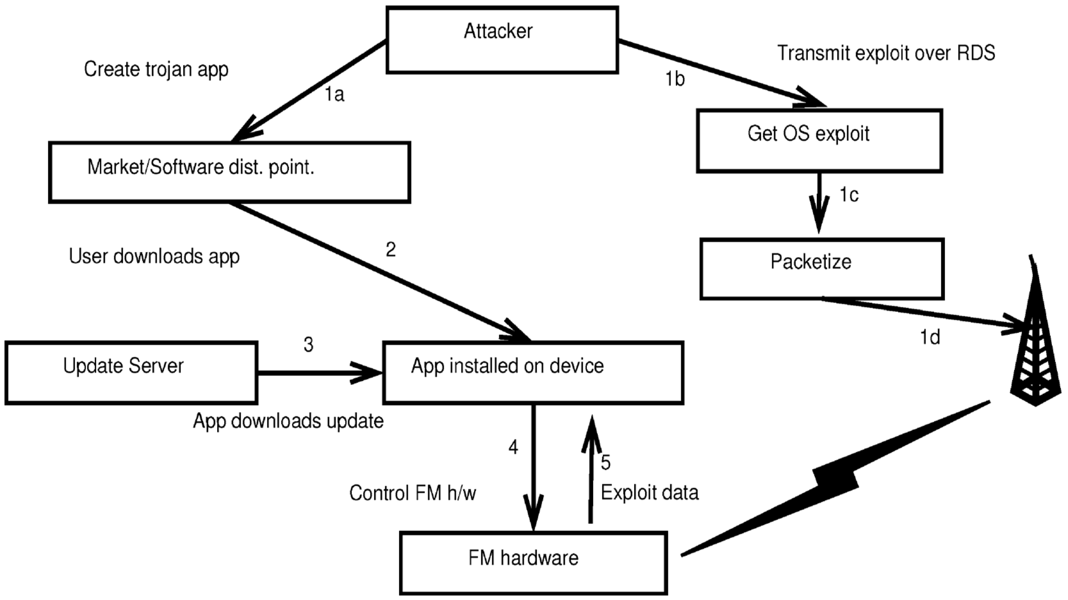

Fig. 2. Attack overview.

method of hijacking analog transmissions wherein a high powered transmitter is situated geographically closer to the intended victims, but after the original transmitter. The effect of this is essentially jamming the original signal and transmitting a totally new signal.

Phase 2: The user downloads and installs the trojan application, thus making the device a target for the attack (Step 2). The app then contacts an update server to update its functionality (Step 3). In the case of an FM player, basic functionality can be only playing of music, and the update could add the ability to record. The side-effect is that the attacker uses this to also push FM hardware control code into the app. This extra code decodes and assembles the payload (Steps 4 and 5). On an Android related note, we achieve this through the recently added API class `DexClassLoader` [5]. This class allows us to plug in code at runtime into a previously installed Android application. This has the advantage of bypassing any static checks implemented in the Android market (Google recently revealed a service code named 'Bouncer' which performs checks on Android market applications [9]). The trojan will appear to be not harmful since the only "suspicious" activity is the download of a class at runtime.

An alternative method to introduce the trojan in the phone could be to leverage the Class Hijacking attack [2]. This attack leverages insecure programming techniques of not adhering to Android secure coding guidelines [21] utilized by the target application: an application uses `DexClassLoader` to load classes from an SD card—this behavior should be avoided, since there is no access control on SD, and the classes to be loaded could be modified by other applications.

Phase 3: Finally, the assembled payload is executed by the trojan to elevate privileges.

C. Alternative/Existing Attack Methodologies

We note that there are existing methods of downloading malicious code into a device. However, we also stress that they are quite different from the RDS attack presented in this paper. Traditional code infiltration techniques include drive-by downloads and downloads of "nonexecutable" data such as JPEG images which contain malicious code fragments embedded in them. An example is the recent drive-by download attack "*NotCompatible*" [11]. The way the drive-by download attack works is that when a website is compromised, the attacker inserts a hidden iframe at the bottom of a webpage. When the user accesses the webpage from his mobile device, the browser automatically starts a download of an APK file specified in the iframe. When the download is complete, the system will prompt the user to install the application. *NotCompatible* disguises itself as a system update, and an unsuspecting user will confirm installation. However, for this attack to be successful, two conditions need to be met. The first is that side-loading of applications needs to be activated and the second is that the user must actually confirm installation. Contrasting this with our attack, the benign app is hosted on the official market and side-loading does not need to be activated. Additionally, the point to note is that in these mechanisms, there exists a commonality of the transmission medium. All utilize the network and content scanning solutions for the network exist. Contrasting this with our RDS attack, no solutions exist that will detect or prevent them.

IV. Specific Attacks

All of the attacks work on Android devices running antivirus software. We have validated the attacks with the help of popular antiviruses. In particular, we installed two free and three paid antiviruses and ran a complete attack (see Section IV-C for an in-depth discussion) on the *Samsung Galaxy S*. The results are summarized in Table I. Traditionally written malware are detected by antiviruses [8], [30]. This is because they embed known exploits directly in the APK, which also causes the package name to get blacklisted. A scanner will detect these signatures since they can scan the APK. However, our malware approach is different in the sense that the application contains

TABLE I
ANTIVIRUSES USED

Antivirus	**Category**	**Exploit**	**Configuration**	**Detected?**
Norton Mobile Security Lite	Free	GingerBreak	Anti-Malware defense activated, Daily scan, SD Card scan	**No**
Lookout Security	Free	GingerBreak	Daily scan, "security" activated, complete scan when malware was installed	**No**
AVG Antivirus Pro	Paid	GingerBreak	Full scan mode	**No**
Kaspersky Mobile Security	Paid	GingerBreak	Full scan with malware existing in binary form in app directory, also `Memory Scan` while malware in main memory, with exploit in binary form in app directory	**No**
AVG Security Pro	Paid	GingerBreak	Full scan mode	**No**

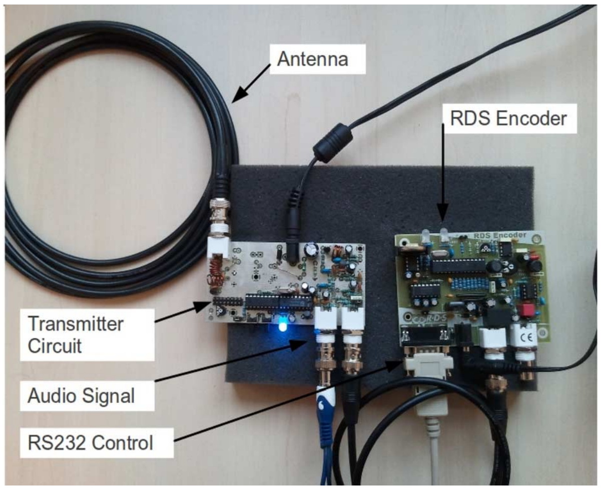

Fig. 3. Transmitter setup.

no exploits, and it receives these exploits over a novel channel. The antiviruses were configured to be in their most secure state. This validation is important as malicious applications assemble and execute privilege escalations from the file system.

We use several low cost off-the-shelf components to build a system that a malicious person could use to mount an attack. Specifically, we use the *PIRA32* RDS generator [12], and a standard FM Transmitter (2 mW). The RDS generator is capable of generating signals that are compliant with the RDS protocol. This is interfaced to a laptop computer via a RS232 to USB adapter. The antenna for the FM Radio can be constructed in several ways. We just use a length of coaxial cable, with an impedance of about 50 ohms.

The *PIRA32* is controlled by an ASCII based instruction set. Fig. 3 depicts the transmitter setup. We have an audio signal over BNC (Bayonet Neill-Concelman is a type of RF connector), as well as the RDS encoder connected over BNC to the transmitter circuit (the circuit to the left). Fig. 4 shows the *Parrot Asteroid* car radio along with the exploit transmitter.

A. Packaging the Exploit

In this section we explain how we leverage the RDS signal to send our exploit to the victim's phone.

From Section II-A, we recall how RDS works. Each data unit is 26 bits in length. Each of these data units can be either of type *Program Service (PS)* or *RadioText* (among others). It takes 87.6 ms to transmit four such data units which is the basic protocol data unit (104 bits in length).

Now, we describe the payload transmission protocol. Say P is the payload to be transmitted (e.g., a file). We first send a PS packet (we remind the reader that the PS packet has, by definition, 8 bytes of payload). We choose to set the first two bytes to be `et` (these are used on the client side to recognize our exploit packets), the remaining 6 bytes are numbers in decimal form (containing the size of the payload—in terms of bytes—that we are going to transmit next). We observe that for our purposes, a value that we can specify with 6 digits, is enough.

Then we send as many RT packets as required such that our payload, P is transmitted completely. In particular:

- P is split into packets of 61 bytes each.
- For each packet we add a three byte sequence number. The sequence number is the ASCII form of decimal numbers.
- After adding the sequence number, each packet will contain 64 bytes.
- Since we can have at most 1000 packets of 61 bytes of payload each, the payload transmitted is at most 61000 bytes. In the case of Android, we would like to point out that this much space is more than enough for all known privilege escalation exploits.
- For each resulting packet, we set its value into the RDS transmitter and then ask the transmitter (with the `RT1` command) to transmit that data (as an RT packet) for a time period of 7 seconds.

B. Alternative Transmission Algorithms

As the FM channel is lossy and packet sizes are relatively small, a discussion of why we choose the above algorithm is needed. One standard method of dealing with packet loss is to request retransmissions of dropped packets. However, this is not applicable to our situation for several reasons. The first reason is that retransmission requires the presence of a feedback channel and FM is unidirectional. To overcome this, one may propose to use the network (cellular or WiFi) to transmit sequence numbers of packets to be retransmitted. This mechanism dampens the stealthiness of our attack and opens it up to detection by traditional network scanning. Another drawback is that it creates a footprint of activity which violates one of the advantages of our attack - negligible footprint. Additionally, multiple retransmission requests introduce high overheads on the side of the transmitter since it needs to serve each request individually. Consider the situation where the attacker is in a stadium with 500,000 people. Assuming a 100,000 of those phones have the benign application, it would generate more than a 100,000

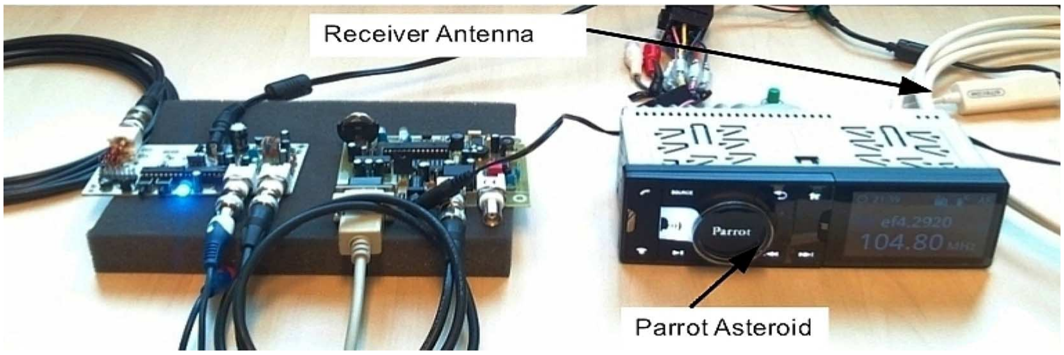

Fig. 4. System setup (devices placed adjacently only for illustrative purposes).

retransmission requests (assuming every infected phone experiences a dropped packet). This will result in the transmitter spending a large majority of its time serving retransmission requests rather than transmitting the payload in a cyclic manner.

C. Attacking Smartphones

Several commercial Android phones in use today already feature an FM Radio with RDS. For our attack, we consider a recent model, the *Samsung Galaxy S* (one of the most popular Samsung phones [13]). It runs the *GingerBread* version of Android. (*GingerBread* runs on 58% of all Android devices [3]). Since there is no public API that allows our client app to control the FM Radio hardware, we reverse engineered the FM Radio subsystem on the *Galaxy S* and obtained full control of the FM hardware, through what is known as a "Stub-Method trick"[1]. In this method, packages (read: stubs) are created with the same name as those of the internal packages we wish to access. This is just to compile the application. These stubs are empty. When the system loads classes for an application with these packages, the loader notices that the system's boot classpath contains the same packages. So, it prevents the loading of the stubs and loads the *actual* classes. Note that an application written in such a way will not experience any problems when it is deployed to the Android market. Thus, we gain access to internal methods. In the case of the FM subsystem, we are able to completely control the FM receiver chip, including powering it on and off as per our discretion. Furthermore, this reverse engineering method is extensible to different FM Radio hardware on different phones and devices as well, provided they conform to a small set of rules. These are as follows:

1) A Java based API is constructed around the FM chip device driver. This can be in the form of a System Server. This is a common design practice on Android. In fact, many of the other hardware devices are controlled in a similar manner (GPS, WiFi).
2) The class library exists in an internal package. Again, this is a common design pattern on Android. Private functionality exists in internal classes until they are mature enough for general use.
3) The class library is not protected by permission strings (see Section II-C). This is crucial, and represents a failure to adhere to Android's secure coding guidelines. Among all the devices we attacked, none of them were protected by permission strings.

1) Reverse Engineering the FM Radio API on the Samsung Galaxy S: The *Samsung Galaxy S* features an *Si4709* FM receiver chip. Samsung has built an internal API for its FM-Radio app. These APIs are hidden and therefore not directly accessible to Android applications written with the standard SDK (Software Development Kit). However, this is yet another example of security by obscurity, since the API exists in the internal `com.samsung.media.fmradio` package and more importantly their access is not protected since they do not use `checkPermission` (which is a standard Android security mechanism) to protect the FM service. As a consequence, a normal app can control the FM chip and receive RDS.

To reverse engineer this internal API we used `ded` decompiler, `smali` and `baksmali` tools. `ded` is a dalvik decompiler [25] for dalvik classes. `smali` and `baksmali` are decompilation tools, in this case used for conversion of `odex` to `dex` and hence to `jar`, as we will see later.

1) We begin by extracting the FM Radio application package, `FMRadio.apk` from the *Samsung Galaxy S*. We then run `ded`, as well as `smali` on this APK. By reading the disassembled code (and cross checking the outputs of the two tools), we were able to know that the application depends on a system provided functionality which is implemented as a system server.
2) We then pull the core framework files, as these will contain the code for the system server. On a device the files are `core.odex`, `framework.odex`, `services.odex`, `ext.odex`, `android.policy.odex`, `core-junit.odex` and `bouncycastle.odex`.
3) After this, we run `baksmali` and `smali` to obtain the normal `dex` format data. We then use `dex2jar` to convert these `dex` format files to the `jar` format. This `jar` file is used to find out the method signatures and library hierarchy.
4) Finally, we use a "stub method trick" to get an Android application to compile. In this method, we create packages and classes with the exact same name as that of the internal methods. When the dalvik class loader tries to load these classes, it sees that our stubs match the internal API. So, it

[1]The steps were also performed on an *HTC Desire HD*, and yet again, we obtained complete and silent access to the FM Radio hardware.

forbids the loading of these stubs, and instead loads the "actual" internal API classes from `framework.jar` which exists on the device.

2) Privilege Escalation With Gingerbreak: As stated in Section III-B, the application downloads a class from a location maintained by the app writer. This class will silently tune to a known frequency and listen for RDS data. Upon detecting a specific byte sequence in the RDS stream, it will start assembling subsequent bytes. The exploit we transmitted is *GingerBreak* [7]. Its size is 7719 bytes (after the packaging step) and its received in 28.3 minutes (this time is for 2 complete transmission cycles. Depending on existing interference, this could reduce, which in turn results in a reception time of 14 minutes). *GingerBreak* is a local privilege escalation attack which makes use of a negative index array write in `vold`, the Android volume daemon. Once the trojan has assembled the payload, it will execute it. The ordinary application (read: trojan) has now elevated to `root` privileges and it can perform any task, ranging from reading private contacts to shutting down the phone.

Twenty eight minutes may seem like a long period of time, however there are many situations in which our phones are exposed for such intervals. Examples can include waiting in the metro station, on the bus or even more interestingly during basketball or football matches, rock concerts and movie theaters. These are situations where hundreds or even thousands of devices are located in a single geographical location for an extended period of time and they can all get infected at the same time.

D. Attacking Car Radios

To demonstrate the ability to deploy malware via RDS across different platforms (and different versions of the Android OS), we attacked a car radio system. We selected the *Parrot Asteroid* car radio that features an *NXP TEF6624* FM receiver chip. The device runs the *Cupcake* version of Android. Also for the Parrot Asteroid we had to reverse engineer the internal FM API following similar steps described in Section IV-C1. We were able to access *RadioText* and *Program Service* name as well as other information like current station information, frequency etc. In our attack, we can just hijack the current frequency and take control of it. Thus, we don't really need to tune to a specific frequency. Otherwise, if the tuner has already been started, we can perform a full scan and use the internal method `gotoFullScanIndex(Long)` to go to a specific frequency.

For the exploit, we have modified the `KillingInTheNameOf` [10] exploit and added some of our own code to enable local `root` access. The new exploit, `KillingCarRadio` is 5328 bytes long (after the packaging step) and we received it over RDS in 19.6 minutes.

1) ADB TCP/IP Exploit for the Parrot Asteroid: We describe a root exploit we run on the *Parrot Asteroid*. Note that a trojan is already installed, which is similar to the one we use for the *Samsung Galaxy S*. There is a facility that allows users to install custom applications on their car radio. The applications are downloaded from a market.

The Android Debug Bridge (ADB) is a debugging daemon for Android devices. It is composed of two components. One is a daemon that runs on the device. This is known as `adbd`. The other component runs on the debugger, or host system. When a developer wishes to debug the device, a connection is established, usually over a USB cable, to the `adbd` daemon running inside the phone. The `adb shell` command opens up an interactive shell on the host. ADB runs as the `shell` user which has more privileges than normal Android applications. It normally waits for USB connections to service debugging requests, however, it can also service these requests over TCP.

On the *Parrot Asteroid*, ADB runs over TCP by default. We connect to ADB from within the device itself, i.e., we masquerade as a host. From the point of view of the ADB daemon, for all purposes, we appear to be a normal debugging host. Using the ADB documentation, `tcpdump` and `tcpick`, we figure out what is the interaction sequence between the ADB host and the ADB daemon. We then write a C socket program which can talk to ADB from within the device. This program will launch `KillingCarRadio` from an ADB shell, and when it eventually is killed (this happens because `KillingCarRadio` requires an ADB restart), we again establish a socket connection to ADB. This time, we get a root shell since ADB is running as root. `KillingCarRadio` is our custom privilege escalation exploit for the Parrot Asteroid. It makes use of a bug in the `ashmem` (Android Shared Memory) implementation (drawing inspiration from `KillingInTheNameOf`), wherein the system property space can be mapped as read/write by any process (note that this entire process is automated. It happens automatically when the binary is executed).

Also, being able to talk to ADB from within the device itself is a privilege escalation to the ADB shell user. We can do several things here. For example, replace the IME (Input Method Extension), or silently install a package. Therefore, its not completely necessary to elevate to root. Sufficient damage can be done by just talking to ADB.

Now that we can elevate to `root` or `shell` on the car radio, we basically "own" the device.

At the moment, our attack can control only the entertainment and the navigation systems provided by the Parrot Asteroid. However, the Parrot Asteroid has a 4-wire UART (Universal Asynchronous Receiver Transmitter) port, and Parrot has announced support for the Steering Wheel Control (SWC) interface soon. This can potentially open up the CAN (Controller Area Network) bus to the attacker and, escalate the type of damage the attacker can achieve by taking direct control of the car electronic control units [22].

E. Attacking Other Platforms

While many smartphones and car radios are equipped with FM Radio receivers many other devices do not have, natively, FM Radio. In this section we describe how FM RDS-based attacks can still affect such devices.

On the market there are several USB dongles supporting FM Radio functionality for less than 50 U.S. dollars. Furthermore, since Android 3.1, a Java based USB Host API has been introduced which makes it possible to write applications that interface with USB devices. We selected one such product, the *ADS/Tech InstantFM Music RDX-155-EF*. It is based on the popular *si470x* chip from SILabs and the associated reference design.

The dongle does not interface with the Android userspace but only Linux, so there is no driver for Android and no way to use this dongle with an APK. However, we wrote such a driver with common APIs. This device exposes the *si470x* chip registers as HID (Human Interface Device) reports.

We have written a driver for the *ADS/Tech FM Receiver* using the USB Host API. We achieve plug and play. The user has to just plug in the dongle, install our application and she instantly receives FM. Actually, offering this functionality might be a good motivation for the people to install our trojan. As a result we can switch on the device, tune to a frequency and assemble *RadioText* and *Program Service Name*, which is exactly what we need to execute an attack. Once this is done, we follow the same procedure as outlined in earlier sections to assemble a malicious payload and execute it to elevate our privileges.

As a demonstration, we successfully mounted an attack on the *Asus EEE Pad Transformer TF101G* tablet which runs the *Honeycomb* version of Android.

Attacks similar to the one described here can potentially be implemented on devices equipped with a USB port and running the Android OS, such as Google-TVs, alarm clocks, e-Readers and digital photo frames. Device users may be tempted to incorporate radio functionality given the ease of carrying out such an integration.

V. Possible Optimizations

There are several possible optimizations to make our attacks even more effective, even if we consider their implementation outside the scope of the paper.

In our attacks, we require approximately 25 minutes (averaged over 5 runs) to receive a payload. Such long time is required also to deal with the inherently lossy nature of the FM channel. A way to reduce the time needed to receive the payload is the use of proposed standards like eXtended RDS (xRDS) [6], which is expected to achieve a data rate of 2 Kbits/s (transmitting up to 6 RDS channels on a single FM carrier).

Apart from these new standards that can become practical in the near future, a simple optimization we already successfully tested is to use multiple frequencies and transmit the different parts of the exploits in parallel. In fact, we observe that the time issue is due to the unidirectional nature of the FM channel (from the station to the receiver). As described in Section IV-A, the approach we used to transmit a (long) data packet is simply to transmit packets repeatedly and consecutively, until all the packets have been received correctly. Since we cannot have a feedback channel from the receiver, our idea is to parallelize rather than serialize the packets to be transmitted, and transmit groups of them on a different frequency. In this way, we reduce the overall transmission time at the cost of more complex receiver logic that should deal with reconstructing the original payload.

VI. Countermeasures

Now that we have demonstrated how to exploit FM Radio broadcasts, it is important to understand how to prevent and fix these problems. Concerned parties can protect their systems against attacks like the ones shown in this paper by taking the following steps:

1) Prevent Unauthorized Reception of Data via the Covert FM Channel: The reader may have noticed that it was possible for us to gain access to FM hardware mainly due to the fact that it was not protected by the appropriate permissions (Section IV-C). Android provides a permission model (Section II-C), and if the guidelines were followed correctly [4], we would not have been able to control the FM hardware silently. Hence, adherence to secure coding guidelines would have prevented the attacker from accessing the data on the FM channel. This is an example of a preventive countermeasure.

2) Prevent an App From Acting on Received Data: This method relies on the ability of antiviruses to work correctly, as the attack mainly uses known OS exploits. The main problem with Android antiviruses is that they have to run as a normal user application due to the Android sandbox. In order for an antivirus or anti-malware product to perform the analysis required to detect malware, there are a few basic requirements that should be provided by the underlying operating system. These are as follows:

1) The ability to intercept packet data on the network interface, i.e., a firewall mechanism.
2) The ability to scan the whole file system and all data contained in it.
3) The ability to scan process memory (virtual memory).

These requirements can be accomplished if the AV product runs with sufficient privileges. On a Linux-like system, this coincides with running with `root` privileges. There is no mechanism to accomplish this on Android. Therefore, we propose changes that will allow "certified" antiviruses to elevate privileges in a controlled fashion. In this way, an antivirus will be able to scan all private memory and storage directories of other applications, and detect whether these applications utilize known exploits. We observe that this is not possible with the current Android sandbox. A possible modification could be the following security model. The antiviruses could be signed by the system key (allowing them to elevate privileges). Companies that wish to develop antiviruses for the Android platform, must undergo a code review process similar to one used for the Apple Store, and get "certified" by the vendor—both figuratively and literally. The Android system might make a special API available for antivirus purposes—to scan memory and private structures. The access to this API will be protected by the signature permission.

Similarly, suppose we have a special process group, say the `av` group: all processes which belong to this group (i.e., all "certified" antiviruses) will be able to use the special API mentioned above. In turn, an antivirus can be assigned to the `av` group at installation time. This is done by the Android installer. Let us assume an application declares to be an antivirus. When its installation is requested via the Android Market, the Market can check with the Certification Authority whether the application is actually authorized. If it is so, the Market will communicate this to the Android installer. Note that this change only applies to antivirus products. For all other products, the deployment cycle is the same as it is now. From the user's point of view, there is no change at all.

Another mechanism is to extend Android's architecture such that it supports the loading of "privileged pluggable modules". These modules are similar in spirit to Linux kernel modules, but their task in the context of Android is to enable more effective Antiviruses. As stated earlier in this section, giving the right privileges to Antiviruses will allow scanning of private memory areas and can potentially detect RDS attacks. One may note that supporting pluggable modules would require a change in the signature scheme that Android uses currently, since we want Antiviruses to be developed by trusted AV vendors and not system developers. However, making this work correctly within the current Android signature scheme is an open research question.

Finally, another approach for a countermeasure is to focus on defending from the malicious FM channel itself. While content certification even if effective, might require a huge change in the FM infrastructure, more practical solutions could be to use techniques like content validation [38], and radio fingerprinting [37].

VII. Related Work

Since our attacks relate to the security of the Internet of Things (IoT) and portable devices in particular, we consider existing work on this issue. While some assessment of IoT security has been done [14], it has been only considered from an architectural point of view, without a complete analysis of possible attack vectors. Nevertheless, there exists previous work investigating some attack vectors for the Internet of Things. As an example, in [38] the authors raise the attention to malware spreading via RFID tags. Other more recent work focused on attack vectors for automobiles [32], [22], which can be seen to some extent as items of the internet of things. In [32], the authors demonstrate that an attacker can circumvent a broad array of safety-critical systems, and control a wide range of automotive functions and completely ignore driver's input, by assuming a malicious component is already present in the system. This latter assumption has been further relaxed in [22], where the authors show that remote exploitation is also possible via a broad range of attack vectors (including mechanics tools, CD players, Bluetooth and cellular radio), hence paving the way for long distance vehicle control, location tracking, in-cabin audio ex-filtration and theft.

Coming to RDS, there exists previous work already suggesting the use of RDS in a malicious way [19], [41]. In particular, in [19], [41], the authors exploit RDS to hijack the TMC (Traffic Message Channel) to spoof RDS-TMC reception, and thereby convincing GPS-based navigators to change the calculated route. However, these attacks target RDS services and do not exploit RDS for malware deployment. While we leverage RDS channel in a malicious way as the work in [19], [41], our approach is completely different. In fact, in [19], [41] authors only exploit the implicit vulnerabilities of the RDS protocol, and only to attack systems which are usually dependent on RDS data (i.e., RDS assisted navigation systems). Instead, in our approach, we just leverage the RDS channel to convey part of the attack data, in a way which is stealthy and undetected by common security mechanisms (e.g., antiviruses)—our target being to attack the end-device and not just a service depending on RDS.

We underline that the proposed attack detection techniques, like the one introduced in [31], would not be effective against our novel approach to stealthily attack any device equipped with RDS capabilities. In fact, in [31] the authors base the detection of malicious behavior on the anomaly of energy consumption, with the main purpose of detecting energy depletion attacks. Firstly, our attack target is not to deplete energy. Furthermore, there is no specific energy consumption pattern associated with our actual attack: the FM Radio is active any time the apparently honest application is running—not only when the FM Radio actually receives the malicious exploit.

Our attacks were possible also due to some limitations of the Android OS to prevent or at least detect them.

After the first version of Android had been published, researchers put a significant effort to discover security vulnerabilities that malicious applications can leverage to thwart the system. Possible countermeasures have also been proposed. One of the first solutions, Kirin [26], aimed at providing a lightweight certification mechanism for applications at the time of installation. Similar security policy enforcement has been also proposed considering the run-time behavior [34], [23]. Other researchers focused on restricting access to data [20], [29]. However, restricting the permission of applications or data accesses might not be enough to resolve the issues raised by malware. In fact, malicious applications can also act by means of other (honest) applications. Some solutions to address this problem have been proposed [24], [39], however they are not backward compatible with most existing applications.

While a significant research effort has been put to limit the effect of malicious mobile applications (in Android, as well as for other systems for mobile devices [28], [33]), little effort has been concentrated on investigating actual attack vectors. Indeed, all the security confinements and restrictions imposed by the described systems become ineffective when the following conditions hold for a malicious application: i) it is not identified as malware by antiviruses or other security systems; and ii) it is able to gain root privileges.

A different approach in detecting malware focuses on offloading heavy computations from the mobile device (e.g., a smartphone, which is resource constrained) to the cloud [35]: the basic idea is to have on the cloud a clone of the actual device; to replicate in the clone all the actions being run in the device; hence, detecting the malicious behavior (which is a computation-demanding activity) on the cloud clone, rather than on the actual device. While this approach might be interesting from the computation-saving point of view, we argue that it still requires a significant amount of communication (with the cloud) plus it adds delay. Another solution to temper malware in Android is SEAndroid [15]. The idea is to replicate the experience of SELinux on the Linux layer of Android. Since the complexity of configuring SELinux was one of the main deterrents to its wide adoption, we foresee similar, if not worse problems with SEAndroid since most smartphone users are not security experts.

VIII. Conclusions and Future Work

After being envisaged by researchers and companies, a world of always on and always connected mobile devices is finally becoming a reality, thanks to the miniaturization and the spread of pervasive services (e.g., cloud). Furthermore, an increasing number of these devices are equipped with FM Radio functionalities, which implies the availability of the FM Radio Data System (RDS) channel.

In this work, we have shown for the first time how to exploit the FM RDS protocol as an attack vector to deploy malware that, when executed, gains full control of the victim's device. In particular, we have shown how this attack vector allows an adversary to deploy malware on different types of devices (smartphones, car radios, and tablets). Furthermore, our attack implementation on the Android OS has shown that the infection is undetected on devices running this OS: malware detection solutions are limited in their ability due to the current design of the Android security model.

As we experienced in our work, setting up an FM transmitter is not only quite simple but also cheap: the economic effort to spread the malware is only a few hundred U.S. dollars. Furthermore, the inherently broadcast nature of the FM Radio makes the vulnerability even more important, making it easy to attack a large number of devices in a given area at once. Finally, the mobile devices which constitute the envisaged Internet of Things are not only currently highly exposed, but can also become the Internet of Bad Things. In fact, many devices are also equipped with FM transmitter capabilities (an example is the *Nokia N900*, which comes with an FM API). Hence, it is easy to see how we can not only affect a large number of devices at once, but also the devices already infected can become malicious FM transmitters, and continue spreading the world's first FM-based worm, and recruiting nodes for the first FM-based botnet.

To the best of our knowledge, as this is the first paper that looks at the FM RDS channel from this point of view, this work opens up a new vein of research, which is needed to improve the security of all the pervasive mobile devices we surround ourselves with.

References

[1] AlertFM, Mississippi Emergency Management Agency (MSEMA) [Online]. Available: http://www.msema.org/documents/AlertFM4.22.10.pdf

[2] Android Class Loading Hijacking. Symantec Security Report [Online]. Available: http://www.symantec.com/connect/blogs/android-class-loading-hijacking

[3] Android Distribution. Google [Online]. Available: http://developer.android.com/resources/dashboard/platform-versions.html

[4] Developing Secure Mobile Applications for Android. Jesse Burns, iSec Partners [Online]. Available: https://isecpartners.com/files/isec_securing_android_apps.pdf

[5] DexClassLoader Android API. Google Android Developers [Online]. Available: http://developer.android.com/reference/dalvik/system/DexClassLoader.html

[6] eXtended RDS protocol. xRDS [Online]. Available: http://www.extended-rds.org/

[7] GingerBreak, Mitre, CVE-2011-1823 [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2011-1823

[8] X. Jiang, GingerMaster Malware [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/

[9] H. Lockheimer, Google reveals the Bouncer service for the Android Market, Google [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[10] KillingInTheNameOf, Mitre, CVE-2011-1149 [Online]. Available: http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2011-1149

[11] NotCompatible Drive-By Download Attack on Android devices. Lookout Security [Online]. Available: https://blog.lookout.com/blog/2012/05/02/security-alert-hacked-websites-serve-suspicious-android-apps-noncompatible/

[12] PIRA32 RDS Encoder module. PIRA CZ [Online]. Available: http://www.pira.cz/rds/pira32.asp?p=PIRA32_RDS_Encoder_Module

[13] "Samsung Galaxy S, the most popular Samsung Phone," *PCMag* [Online]. Available: http://www.pcmag.com/article2/0,2817,2394782,00.asp

[14] O. Garcia-Morchon *et al.*, Security Considerations in the IP-based Internet of Things [Online]. Available: http://tools.ietf.org/html/draft-garcia-coresecurity-03 Internet Draft

[15] S. Smalley, The Case for SE Android, National Security Agency. SELinux Project [Online]. Available: http://selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf

[16] The "Max Headroom" Incident [Online]. Available: http://winstonengle.tripod.com/chicagowho/maxhead.htm

[17] Android on the Boeing 787, Engadget, 2011 [Online]. Available: http://www.engadget.com/2011/09/16/boeing-hitches-android-to-its-787-dreamliner-ridepowers-in-fli/

[18] Congressional Mandate for FM Receivers in portable electronics. ArsTechnica [Online]. Available: http://arstechnica.com/tech-policy/news/2010/08/radio-riaa-mandatory-fm-radio-in-cell-phones-is-the-future.ars

[19] A. Barisani and D. Bianco, "Hijacking RDS TMC traffic information signal," *Phrack Mag.* 2011 [Online]. Available: http://www.phrack.org/issues.html?issue=64&id=5

[20] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, 2011, pp. 51–62.

[21] J. Burns, Developing Secure Mobile Applications for Android. iSec-Partners [Online]. Available: http://isecpartners.com/files/iSEC_Securing_Android_Apps.pdf

[22] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *Proc. 20th USENIX Conf. Security (SEC'11)*, 2011, pp. 6–6.

[23] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "CRêPE: A system for enforcing fine-grained context-related policies on android," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 5, pp. 1426–1438, Oct. 2012.

[24] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. 20th USENIX Security Symp. (SEC'11)*, San Francisco, CA, USA.

[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. 20th USENIX Security Symp. (SEC'11)*, San Francisco, CA, USA.

[26] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conf. Computer and Communications Security (CCS'09)*, 2009, pp. 235–245.

[27] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, Jan./Feb. 2009.

[28] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, 2011, pp. 3–14.

[29] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retroffiting android to protect data from imperious applications," in *Proc. 18th ACM Conf. Computer and Communications Security (CCS'11)*, 2011, pp. 639–652.

[30] X. Jiang, DroidKungFu3 Malware [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/

[31] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proc. 6th Int. Conf. Mobile Systems, Applications, and Services (MobiSys'08)*, New York, NY, USA, 2008, pp. 239–252, ACM.

[32] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Proc. 2010 IEEE Symp. Security and Privacy (SP'10)*, 2010, pp. 447–462.

[33] C. Miller, "Mobile attacks and defense," *IEEE Security and Privacy*, vol. 9, no. 4, pp. 68–70, Jul./Aug. 2011.

[34] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proc. 2009 Ann. Computer Security Applications Conf. (ACSAC'09)*, pp. 340–349.

[35] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proc. 26th Ann. Computer Security Applications Conf. (ACSAC'10)*, pp. 347–356.

[36] A. Rahmati, L. Zhong, V. Vasudevan, J. Wickramasuriya, and D. Stewart, "Enabling pervasive mobile applications with the fm radio broadcast data system," in *Proc. Eleventh Workshop on Mobile Computing Systems & Applications (HotMobile'10)*, 2010, pp. 78–83.
[37] K. B. Rasmussen and S. Capkun, "Implications of radio fingerprinting on the security of sensor networks," in *Proc. 3rd Int. Conf. Security and Privacy in Communication Networks (SecureComm'07)*, pp. 331–340.
[38] M. R. Rieback, B. Crispo, and A. S. Tanenbaum, "Is your cat infected with a computer virus?," in *Proc. Fourth Ann. IEEE Int. Conf. Pervasive Computing and Communications (PERCOM'06)*, pp. 169–179.
[39] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Yaase: Yet another android security extension," in *Proc. Third IEEE Int. Conf. Privacy, Security, Risk and Trust (PASSAT'11)*, 2011, pp. 1033–1040.
[40] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *IEEE Security and Privacy*, vol. 8, no. 2, pp. 35–44, Mar./Apr. 2010.
[41] D. Symeonidis, "Rds-tmc spoofing using gnu radio," in *Proc. 6th Karlsruhe Workshop on Software Radios*, 2010, pp. 87–92.
[42] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. Second ACM Conf. Data and Application Security and Privacy (CODASPY'12)*, 2012, pp. 317–326.

Earlence Fernandes graduated with the Bachelor degree in computer engineering from the University of Pune, in 2009. He worked as a Scientific Programmer in the Systems Security group at Vrije Universiteit Amsterdam until 2012. Currently, he is a Ph.D. student at the University of Michigan, Ann Arbor, MI, USA.

Bruno Crispo (S'97–M'99–SM'09) received the M.Sc. degree in computer science from University of Turin, Italy, and the Ph.D. degree in computer science from the University of Cambridge, U.K.

He is an Associate Professor at the University of Trento, Italy. His research interests include networks and distributed systems security, mobile malware analysis and detection, access control, applied cryptography, and security protocols. He is a member of ACM.

Mauro Conti (S'07–M'08) received the Ph.D. degree from Sapienza University of Rome, Italy, in 2009.

In 2008, he was a Visiting Researcher at the Center for Secure Information Systems, George Mason University, Fairfax, VA, USA. After earning his Ph.D. degree, he was a Postdoctoral Researcher at Vrije Universiteit Amsterdam, The Netherlands. From 2011, he is an Assistant Professor at the University of Padua, Italy. In 2012, he was a Visiting Assistant Professor at the University of California, Irvine, CA, USA. His main research interest is in the area of security and privacy. In this area, he has published 50+ papers in international peer-reviewed journals and conferences.

Dr. Conti was a Panelist at ACM CODASPY 2011, and General Chair for SecureComm 2012 and ACM SACMAT 2013. In 2012, he has been awarded by the European Commission with a Marie Curie Fellowship.