From Smashed Screens to Smashed Stacks: Attacking Mobile Phones Using Malicious Aftermarket Parts

Omer Shwartz, Guy Shitrit, Asaf Shabtai, Yossi Oren

Ben-Gurion University
Beer-Sheva, Israel
omershv@post.bgu.ac.il, shitritg@post.bgu.ac.il, shabtaia@bgu.ac.il, yos@bgu.ac.il

***Abstract*—In this preliminary study we present the first practical attack on a modern smartphone which is mounted through a malicious aftermarket replacement part (specifically, a replacement touchscreen). Our attack exploits the lax security checks on the packets traveling between the touchscreen's embedded controller and the phone's main CPU, and is able to achieve kernel-level code execution privileges on modern Android phones protected by SELinux. This attack is memory independent and survives data wipes and factory resets. We evaluate two phones from major vendors and present a proof-of-concept attack in actual hardware on one phone and an emulation level attack on the other. Through a semi-automated source code review of 26 recent Android phones from 8 different vendors, we believe that our attack vector can be applied to many other phones, and that it is very difficult to protect against. Similar attacks should also be possible on other smart devices such as printers, cameras and cars, which similarly contain user-replaceable sub-units.**

1. Introduction

Consumer devices often have components which can be replaced by the user or by third-party service centers. These components are generally called field-replaceable units, or FRUs. Examples of devices with FRUs include interface cards for routers; touch screen, battery and sensor assemblies for mobile phones; ink cartridges for printers; batteries for health sensors; and so on. These replaceable units typically have their own microprocessors and code spaces, and use a very strictly defined hardware or software API to communicate with the device's main secure CPU, as shown in Figure 1. Most sensors and hardware peripherals communicate over simple interfaces such as I^2C (Inter Integrated Chip) and SPI (Serial Peripheral Interface), these interfaces are very lightweight and offer no embedded authentication mechanisms or error detection capabilities [1].

Programs running on a smart device's main CPU interface with the FRUs using device drivers. These drivers often run in privileged mode with kernel-level permissions. In most cases, FRUs are manufactured by third-party original equipment manufacturers (OEMs). Thus, the source code for FRU drivers is supplied by the OEMs to the phone manufacturers, and the phone manufacturers then proceed to integrate this code into their own source code, making minor adjustments to account for minute differences between phone models such as memory locations, I/O bus identifiers, etc. As we show in Section 3, these minor tweaks and modifications make the process of creating and deploying patches for these drivers a very difficult endeavor.

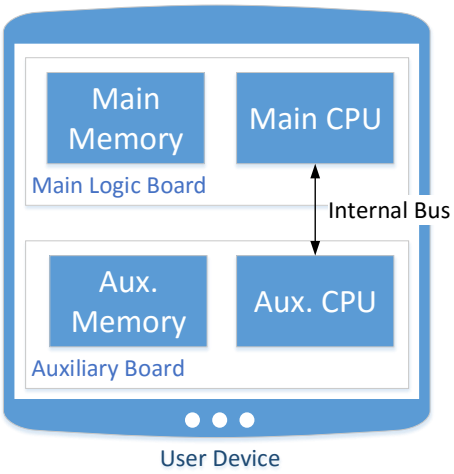

Figure 1. A user device with a field-replaceable accessory board.

In contrast to network or USB drivers, FRU drivers are typically written under the assumption that an authentic, fully trusted hardware component is present on the FRU side. As a result, very few integrity checks are performed on the communication between the FRU's embedded controller and the phone's main CPU. We aim to show that the assumption of a benign FRU is a very risky one to make.

In the remainder of this paper we focus our discussion on replacement touch screen assemblies. According to a global survey carried out in 2015 by Motorola Mo-

bility, around 21% of global smartphone users currently have a cracked or shattered phone screen [2]. Assuming a very conservative estimate of two billion smartphones in the world, this means there are more than 400 million phones worldwide with a cracked screen. For comparison, the Mariposa botnet, which was the largest ever recorded, consisted of about 10 million computers [3]. While some users replace their phone screens using authentic hardware at the original vendors' laboratories, most touchscreen replacements are performed in third-party repair shops. These repair shops often use the cheapest possible screens and thus often provide, knowingly or unknowingly, counterfeit or unbranded subunits. As recently shown by UL [4], a large proportion of presumably authentic replacement hardware purchased on third-party marketplaces such as Amazon or eBay is counterfeit.

In our attack model, we assume that a user has replaced her phone's touchscreen with a counterfeit, malicious component. Such component may include tampered firmware or an embedded malicious IC. Quite trivially, this exposes the user to the risk of key-logging or malicious impersonation. However, as we show in this paper, this counterfeit component can also abuse the CPU-FRU communication bus and achieve kernel-level code execution privileges on the phone.

Our contributions are as follows: we show a proof-of-concept of a physical environment with stock firmware being coerced into running code from an arbitrary memory address; we also show an end-to-end attack in a physical environment with modified software allowing an app to gain root privileges and disable security measures; and we present an analysis highlighting the difficulty in protecting today's devices.

2. The Attack

The objective of our attack was to cause a stock phone running unmodified software to execute arbitrary code by sending malicious inputs through its touchscreen interface. Our initial analysis used a high-end Android phone currently on sale by a major US vendor[1].

2.1. Attack Setup

Our attack setup consists of two parts, a software assisted environment where a full end-to-end attack is shown and a pure physical setup where a preliminary attack is achieved.

2.1.1. Software Assisted Environment. For the software-assisted hardware proof-of-concept, we used normal production configuration used in the stock firmware present on the device, based on Android 6.0.1.

1. The phone's vendor and model, as well as full details on the chain of compromises we used to attack it, will be disclosed after the conclusion of a responsible disclosure process.

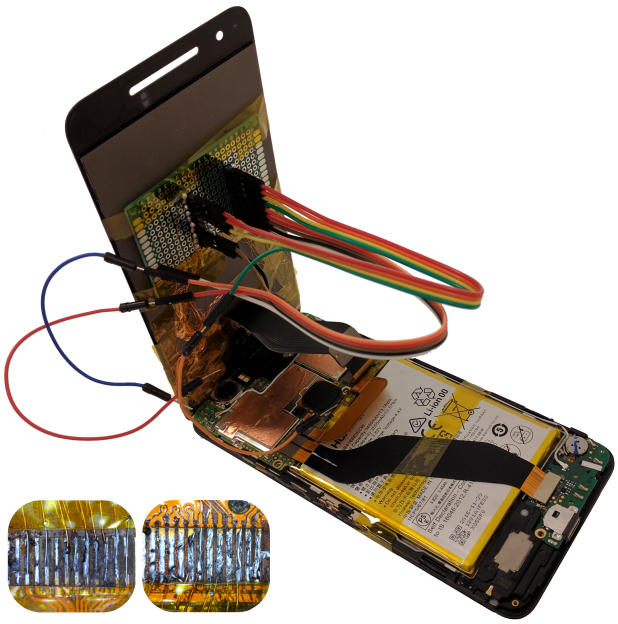

Figure 2. Exposing the mobile phone's touchscreen interface. Inset: wires soldered onto the touch controller communication connection.

We then modified the kernel so that it includes overloaded functions that replace the communication API between the touchscreen driver and the touchscreen. These functions allow intercepting and injecting of packets in the scope of the driver-touchscreen interface. No other parts of the kernel were modified, allowing us to mimic a tampered touchscreen module with minimal kernel modifications. This setup was then tested both in emulation and on the actual phone hardware.

2.1.2. Pure Physical setup. For the full hardware proof-of-concept, an identical phone running stock firmware was disassembled, and the touchscreen controller communication bus was made accessible, as shown in Figure 2. A programmable microcontroller was then placed on the communication bus between the touchscreen and the CPU, allowing custom-crafted responses to be sent to the CPU.

2.2. Preliminary Results

Using our software-assisted hardware setup, we were able to cause several driver software faults which introduced a critical vulnerability into the Android kernel. The vulnerability discovered was exploited using a Return Oriented Programming (ROP) chain (Fig. 3) designed for the ARM64 architecture. This exploitation chain was used to cause a second vulnerability in a user-facing interface. The next step in the attack involved launching an auxiliary user-mode app with no special permissions. This app was capable of continuing the attack by exploiting the new vulnerability via a system call

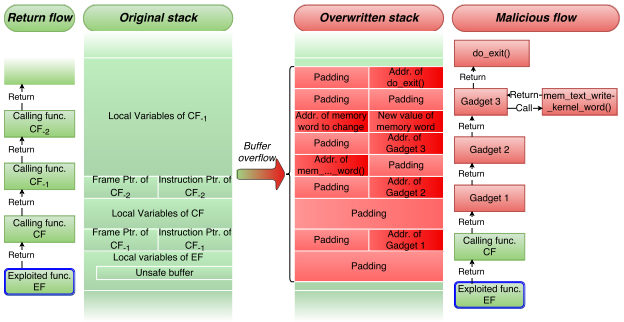

Figure 3. Depiction of the ROP chain resulting in modification of kernel write-protected memory. Gadgets one, two, and three load predefined memory addresses and values from the stack and result in a function call to mem_text_write_kernel_word(), a function that writes over protected kernel memory.

that is normally benign, with the end result of gaining root privileges while disabling the SELinux protection mechanism. Note that our attack model, which assumes a compromised touchscreen, explicitly allows the attacker to perform unauthenticated actions on behalf of the user.

Using our full hardware-based setup, it was possible to send the custom crafted corrupted data during the boot process while injecting data that exploits two vulnerabilities within the device driver. This caused a kernel thread to execute an arbitrary memory address and crash.

3. Driver and Peripheral Diversity in the Android Mobile Phone Landscape

Since our attack used a combination of first-party and OEM code, we were interested in testing the specificity of these vulnerabilities to specific phone models. In order to investigate this question systematically, we produced a driver-level inventory of several popular handsets, and observed the diversity of today's peripheral landscape.

As a consequence of the open-source public license used for Android and for some of its subcomponents, phone vendors are required to release the source code for various Android phones they produce. We downloaded and reviewed the source code of 26 recent Android phones from eight vendors and mapped the features and drivers employed by the devices. This allowed us to observe the diversity and uniqueness of the drivers used as well as assess the difficulty in auditing and maintaining their security. For each phone, the most recent version of the kernel source code was downloaded from the manufacturers website. The default configuration was located and loaded into a script based on kconfiglib.py [5] and the functional drivers were listed.

The actions above allowed us to determine the vendors of the drivers and compare drivers across phones in the purpose of finding identical versions. As shown in Table 1 the output of our survey revealed a vast variance of peripheral vendors across the inspected devices. Of the 89 different drivers we evaluated, only three were used in two or more phone models, and only two were used on three or more phone models. Most of the drivers were unique to their respective device and only a handful of drivers identical among different devices were found.

Phone model	Touch-screen	NFC	Charger IC	Battery	Wire-less Charger
HTC Desire 626	Syn_t1, Atm_t1	NXP_n1	Sum_c1	Max_b1	—
HTC One A9	Syn_t2	NXP_n2	Sum_c2	HTC_b1	—
HTC One M9+	Syn_t3 Max_t1	NXP_n3	Sum_c3	HTC_b2	—
Huawei Honor 5X	Syn_t4	NXP_n4	Hua_c1	TI_b1	—
Huawei Nexus 6P	Syn_t5	$*NXP_n5$	Qua_c1	Qua_b1	—
Lenovo K3 Note	Med_t1	—	Med_c1	Med_b1	—
Lenovo K5 Note	Med_t2	Qua_n1	Sum_c4	Med_b2	—
Lenovo Vibe K4 Note	Med_t3	Med_n1	Tex_c1	Med_b3	—
LG Nexus 4 E960	Syn_t6	Bro_n1	Int_c1	TI_b2	TI_w1
LG Nexus 5	Syn_t7	Bro_n2	Sum_c5	TI_b3	TI_w2
LG G3	Atm_t2	NXP_n6	Sum_c6	$*TI_b4$, Max_b2	TI_w3
LG G4	Syn_t8	NXP_n7	Sum_c7	Max_b3	—
LG Nexus 5X	Syn_t9	$*NXP_n5$	Qua_c2	Qua_b2	—
Motorola Nexus 6	Atm_t3	Bro_n3	Sum_c8	TI_b5, Max_b4	Mot_w1
Samsung Galaxy S5	Atm_t4	NXP_n8, Bro_n4	Sum_c9	TI_b6, Max_b5	—
Samsung Galaxy S6	$*STM_t1$	Sam_n1	$*Max_c1$	$*Max_b6$	Tex_w4
Samsung Galaxy S6 edge	$*STM_t1$	Sam_n2	$*Max_c1$	$*Max_b6$	Tex_w5
Samsung Galaxy S6 edge+	STM_t2	Sam_n3	$*Max_c1$	$*Max_b6$	Tex_w6
Samsung Galaxy S7	Sam_t1	NXP_n9, Sam_n4	Max_c2	Max_b7	IDT_w1
Samsung Galaxy S7 edge	Sam_t2	NXP_n10, Sam_n5	Max_c3	Max_b8	IDT_w2
Sony Xperia Z2	Max_t2	NXP_n11	Qua_c3	$*TI_b4$, Max_b9	—
Sony Xperia M4 Aqua	Syn_t10	NXP_n12	Qua_c4	Sum_b1	—
Sony Xperia M5	Syn_t11	NXP_n13	Med_c2	Med_b4	—
Sony Xperia Z5	Syn_t12, Atm_t5	NXP_n14	Sum_c10	TI_b7	—
Sony Xperia E5	Med_t4	Med_n2	Med_c3	Med_b5	—
Sony Xperia XA	Med_t5	Med_n3	Qno_c1	Med_b6	—

Table 1. Driver survey. Digits indicate unique version of driver. Drivers shared by multiple phones are marked by *. Vendors: Mot: Motorola Mobility; Med: MediaTek; Bro: Broadcom; TI: Texas Instruments; Sam: Samsung; Qno: Qnovo; Hua: Huawei; Int: Intersil; Sum: Summit Microelectronics; Qua: Qualcomm; Syn: Synaptics; STM: STMicroelectronics; Atm: Atmel; Max: Maxim Integrated.

4. Discussion

We showed how a malicious payload delivered by a touchscreen can let an unprivileged app obtain unlimited access to system resources and information on a smartphone. In this section we will discuss the ramifications of such attacks and review both previously explored and unexplored defenses against threats of this kind.

4.1. Analysis of the Attack Surface

There are several unique advantages to this attack method, when compared to traditional malware. First, the attack can be carried out on a phone running a stock operating system, and without having the user perform any action. This makes it difficult to protect against the attack by corporate or carrier-level security policies, or by user education. Second, the attack is performed in a way that leaves no persistent memory footprints. Thus, it is impossible for the kernel or for any antivirus software to detect the malicious activity. Third, since the

attack is located outside the phone's standard storage, it can survive phone factory resets, remote wipes, and firmware updates.

Attacks based on malicious hardware can be divided into two different classes. *First-order attacks* interact with the phone in the ways a standard user would, but without the user's consent. In the case of a malicious touchscreen, the malicious peripheral may log the user's touch activity, or impersonate user touch events in order to call a premium phone number or install malware. *Second order attacks* go beyond exchanging properly-formed data, and attempt to cause a malfunction in the device driver and compromise the operating system kernel. While the results shown here present a successful second order attack, it is still interesting to investigate first order methods on their own, and especially interesting to investigate the combination of both first order and second order attack, in which simulated user interaction is used to enhance a kernel exploit.

The reality we discovered, where every phone model contains a unique driver/version combinations, paints a problematic picture. With thousands of smartphone varieties being used by customers all over the world, maintaining their security is a daunting task.

4.2. Possible Countermeasures

Devices today are being protected from malicious USB or Ethernet connections by means of encryption, blocking, authentication and active detection of suspicious traffic. Such methods can be modified and used on other protocols for communication with device peripherals. Due to the dynamic evolvement of smartphones and smartphone drivers, possible countermeasures should be as generic and adaptive as possible. Preventive countermeasures may include signature verification of the peripheral and related drivers, e.g. using certificates. Detective countermeasures can be in the form of an intrusion detection component that analyzes the communication between the replaceable units and drivers in a trusted manner and detects anomalies.

4.3. Related Work

Smartphones have been a growing target for malicious attacks in the past decade. Some of these attacks are caused by adversaries who abuse the permissions they are explicitly or implicitly given by the user. Attackers can also use various software flaws to maliciously increase their permission level, and thus increase the amount of damage they are capable of causing. These attacks usually result in money theft, exfiltration of private data, aggressive advertising and fraud [6]. Tools and methods for counteracting existing and upcoming malware are being constantly developed and reviewed [7], [8], [6]. While there is a growing focus recently on on the hardware aspect of smartphone security, most security papers still consider the smartphone's hardware as inviolate and are only examining malware-based attacks.

One well-known hardware interface for attacks is the Universal Serial Bus (USB). USB is a widespread standard defining the physical interface and protocols for communication between electronic devices over multiple profiles. The versatility of USB requires careful implementation and broad restrictions against abuse and attacks. Several vulnerabilities have been found in the past that affect Android based phones and allow an attacker to use USB for gaining privileged access to a vulnerable device [9]. Android security patches are released on a monthly basis. A review of 326 Android CVEs (Common Vulnerabilities and Exposures) patched within the last 18 months shows that at least 22.7% (74 items) take place in driver context [10].

The concept of exploiting and defending internal hardware interfaces has been discussed before in the context of industrial applications on common interfaces such as I^2C, and solutions have been suggested [11]. Today's reality of replaceable components exposes and compromises those internal interfaces that were once protected by being a part of a non-serviceable system.

To the best of our knowledge, ours is the first attempt to attack portable devices by means of a lab replaceable malicious hardware component.

5. Conclusions and Future Work

The diversity that exists in the mobile phone peripheral landscape, combined with the high potential for damage, both via first-order attacks and via second-order attacks, is a serious concern which has not been addressed sufficiently by the security community. After proving the feasibility of kernel exploitation through hardware communication channels with replaceable peripherals we intend to complete the demonstration with a pure hardware attack vector. A device will be designed that when attached to a smartphone touchscreen controller is capable of committing an attack on the driver and exploiting a stock device kernel. Further analyses will be performed on additional phone models and their vulnerabilities assessed. It is our intention to also investigate possible countermeasures for such attacks and compile a set of conclusions and designs that will help protect both current and future devices. The rich set of countermeasures which already exists for securing untrusted interfaces such as network and USB should also be considered for addressing this problem domain.

References

[1] NXP, *I2C-bus specification and user manual*, Apr. 2014. [Online]. Available: http://www.nxp.com/documents/user_manual/UM10204.pdf

[2] Motorola Mobility, "Cracked screens and broken hearts - the 2015 motorola global shattered screen survey." [Online]. Available: https://community.motorola.com/blog/cracked-screens-and-broken-hearts

[3] Industrial Control Systems Cyber Emergency Response Team, "Advisory (icsa-10-090-01) - mariposa botnet." [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-10-090-01

[4] UL, "Counterfeit iphone adapters." [Online]. Available: http://library.ul.com/?document=counterfeit-iphone-adapters

[5] U. Magnusson, *Kconfiglib - A flexible Python 2/3 Kconfig parser and library*. [Online]. Available: https://github.com/ulfalizer/Kconfiglib

[6] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[7] A. Arabo and B. Pranggono, "Mobile malware and smart device security: Trends, challenges and solutions," in *2013 19th International Conference on Control Systems and Computer Science*. IEEE, 2013, pp. 526–531.

[8] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[9] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending against malicious peripherals with cinch," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association.

[10] Google, *Android Security Bulletin*. [Online]. Available: https://source.android.com/security/bulletin

[11] J. Lázaro, A. Astarloa, A. Zuloaga, U. Bidarte, and J. Jiménez, "I2csec: A secure serial chip-to-chip communication protocol," *J. Syst. Archit.*, vol. 57, no. 2, pp. 206–213, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2010.12.001