Two Vulnerabilities in Android OS Kernel

Xiali Hei, Xiaojiang Du and Shan Lin
Department of Computer and Information Sciences
Temple University
Philadelphia, PA 19122, USA
Email: {xiali.hei, dux, shan.lin}@temple.edu

***Abstract*—Android Honeycomb operating system is widely used for tablet devices, such as Samsung Galaxy Tab. The Android system programs are usually efficient and secure in memory management. However, there has been a few security issues reported that show Android's insufficient protection to the kernel. In this work, we reveal a new security pitfall in memory management that can cause severe errors and even system failures. Existing security software for android do not detect this pitfall, due to the private implementation of Android kernel. We then discuss two vulnerabilities introduced by this pitfall: 1) malicious programs can escalate the root-level privilege of a process, through which it can disable the security software, implant malicious codes and install rootkits in the kernel; 2) deny of service attacks can be launched. Experiments have been conducted to verify these two vulnerabilities on Samsung Galaxy Tab 10.1 with Tegra 2 CPU. To protect systems from these vulnerabilities, we proposed a patching solution, which has been adopted by Google.**

***Index Terms*—Android Honeycomb OS; Kernel privileges elevating; DoS; Nvidia Tegra**

I. Introduction

As the mobile computing technology advances, the Linux-based Android operating system specially designed for touch screen mobile devices are becoming more and more popular. International Data Corporation [2] believes that Android will maintain its overall leadership position in mobile device market throughout 2016, but competition among BlackBerry, iOS, and Windows Phone will shift position each year. Secure and reliable Android operating system is critical for its success.

A central principle of the Android security architecture is that no application, by default, has permission to perform any operation that would adversely impact other applications, the operating system, or the user [24]. To meet this principle, Android sandboxes each application by combining Virtual Machines together with the Linux access control. These two mechanisms are well studied to achieve a high level of security. Basically, each application is considered as an individual Linux user. However, the Linux kernel that Android built upon may still hide unchecked vulnerabilities. Such kernel level vulnerabilities could be fatal.

We reveal a previously unknown pitfall in the Tegra 2 CPU driver of the Android Honeycomb operating system. All of the early dual-core Android devices were running on Nvidia's Tegra 2 platform, such as Samsung Glaxy Tab, LTE Galaxy Tab 10.1, and Motorola Atrix and Droid X2 [1], etc.. Using this pitfall, malicious memory access can be executed for system breach. In particular, we present two new vulnerabilities raised by this security issue in Android OS: 1) a malicious user can overwrite a system parameter to escalate his privilege to the root level, then he can disable the antivirus-software and deploy malware and other programs to collect critical system users' information. 2) a malicious user can force the system to shut down unexpectedly, and even launch a Denial-of-Service (DoS) attack. The DoS attack can make the device completely unresponsive. These vulnerabilities potentially can be exploited remotely. In the worst case, thousands of Android devices could be affected.

Android requires each app to explicitly request permissions before accessing personal information and phone features. The requested permissions allow a user to evaluate the app's capability and determine whether or not to install the app firstly. Due to the dominant role of the permission-based model in running Android apps, it is critical that this model is properly enforced in existing Android smart phones. If malicious applications exploit the two vulnerabilities in our paper, all the users run this kind of malicious applications will be out of service.

To overcome these two vulnerabilities, we propose a solution that requires small changes in the Android OS. We have reported these two vulnerabilities to Google. Google has verified and accepted them. Moreover, our proposed solution has been adopted; and a security patch will be published to fix the problem.

We conducted experiments on Android Honeycomb 3.1 using the Samsung Galaxy Tab 10.1 with Nvidia Tegra CPU. And the results show that we can easily exploit these vulnerabilities and we can solve them with our fix methods.

The contributions of our work is summarized as follows:

- We revealed a security pitfall in the Tegra 2 CPU driver program on the Android operating system. A couple severe security vulnerabilities are exposed by exploiting this pitfall.
- We demonstrated how to perform system privilege escalation and denial-of-service attack using real Samsung Galaxy Tablet.
- We proposed a solution to fix the pitfall, our report to the problem has been accepted by Google.

The rest of the paper is organized as follows. In Section II we provide a brief background introduction on Android OS. In Section III, we discuss two identified vulnerabilities in Android kernel. In Section IV we illustrate two solutions respectively. In Section V we present real system experimental

results that verify the two vulnerabilities and their solutions. In Section VI we investigate works related to this work. We conclude the paper in Section VII with some final remarks.

II. Background

A. The Android Architecture

The Android Architecture consists of 5 layers. Other than the Linux kernel at the bottom layer, there are four Android-specific layers. From top to bottom, they are the Application layer, the Application Framework layer, the Android Runtime layer, and the Libraries layer. In this study, we focus on the Linux kernel layer of the system. We also briefly introduce the Android developer bridge and the Nvidia Tegra 2 in this section. The Android developer bridge allows a basic physical attack approach to the system. And the Nvidia Tegra 2 CPU driver in Linux kernel layer is where we found the security pitfall.

B. The Linux Kernel Layer

Android Honeycomb relies on Linux kernel version 2.6.36 for core system services, such as process management, inter process communication, security service, memory management, network stack, and driver models. Linux is a macro-kernel operating system. The CPU and IO drivers are located within the kernel (the same address space) for high efficiency. However, the kernel is exposed to more security risks, especially when kernel components, like drivers, have security issues.

The hardware abstraction layer (HAL) library works on the kernel and interact with the Linux kernel through system calls. Some of the system calls exposed by the Motorola Droid Bionic are normally not available to user space because they're excluded by the use of the #ifdef _KERNEL_ and #endif guards. By including a system call in its new C library, Android can define any system call to the kernel from the user space as a "normal system call". So many kernel function are exposed to Android users.

C. Android Developer Bridge (ADB)

Android Debug Bridge (ADB) is a command line tool that allows your local computer to communicate with an connected Android-powered device or an emulator. It is a client-server program that includes a client, a daemon, and a server. ADB makes a connection between your telephone or other personal wireless devices and a local computer, creating the possibility to interact with your telephone or tablets on your desktop through the command line. An attacker can obtain privileged access through physical access to a device that has ADB enabled [25]. If the attacker can access the physical computer, he/her can easily determine whether ADB is enabled or not by executing *adb get-serialno* on the computer. The device's serial number would be returned if the ADB is enabled. Once the attacker knows that ADB is enabled on the device, he can use ADB's *push* command to implant an exploit on the device, and use ADB's *shell* command to launch the exploit and escalate his privilege.

An attack on an ADB enabled device does not require any action from the user and it is more cleaner compared with remote attacks. Privilege escalation using ADB has a drawback that depends on the availability of an enabled debug bridge. However, if the device is not password-protected, the attacker could simply connect with the common device interface and enable ADB. For instance, Super One-Click desktop application in paper [27] can gain privileged access from Android devices with enabled ADB and give the user privileged access. ADB-based attacks do not need install new application and reboot.

The lack of device modification in ADB-based attacks makes it much more difficult to trace than other attacks. It is unlikely to be detected by security applications on unrooted devices.

D. Nvidia Tegra 2

Tegra developed by Nvidia is a system on a chip (SoC) series using ARM architecture processor CPU and GPU for mobile devices such as smart phones, personal digital assistants, and mobile Internet devices. Specially, it emphasizes low power consumption and high performance for playing audio and video. Tegra 2 is the world's first mobile dual-core CPU, which integrated ARM Cortex-A9 and allowed a out-of-order execution for more efficient processing and better overall performance.

III. The Two Vulnerabilities

We examine the source codes of two packages: GT-P7500_OpenSource.zip and GT-P7510_OpenSource.zip [28], and we find two vulnerabilities in the nvhost_ioctl_ctrl_module_regrdwr function in the file dev.c.

The *nvhost_ioctl_ctrl_module_regrdwr* function has two sub-functions: *nvhost_write_module_regs* and *nvhost_read_module_regs*. The first vulnerability is in the nvhost_write_module_regs sub-function. The *Get_user(offs, offsets)* in Line 561 is used to get the offset from users. The *nvhost_write_module_regs(&ctx->dev->cpuaccess, args->id, offs, batch, vals)* in Line 569 determines the location based on offs from users (the offs are used as offset to write in registers). Because there is no boundary check on "offs", it creates a kernel buffer overflow vulnerability that allows arbitrary memory access. Hackers can exploit the vulnerability to escalate kernel privileges. The *nvhost_read_module_regs* sub-function has similar vulnerability.

After exploiting the vulnerability, we have full control of the Android device, i.e., as a root user in the shell. We can access the kernel logs during the running of a fuzzy test. We write the source code of the fuzzy test by ourselves. By analyzing kernel logs during that period, we find the second vulnerability, which is in Line 598: BUG_ON(_IOC_SIZE(cmd)->NVHOST_IOCTL_CTRL_MAX_ARG_SIZE). The program fails to check the size of _IOC_SIZE(cmd), and this can cause a DoS attack to crash the operation system.

```
<4>[  204.876828] !@Sync 6
<1>[  211.022744] Unable to handle kernel paging request at virtual address 06900000
<1>[  211.030614] pgd = d8850000
<1>[  211.034037] [06900000] *pgd=00000000
<0>[  211.038295] Internal error: Oops: 5 [#1] PREEMPT SMP
<0>[  211.043870] last sysfs file: /sys/devices/virtual/sec/sec_misc/usbsel
<4>[  211.050644] Modules linked in: dhd
<4>[  211.055332] CPU: 1    Not tainted  (2.6.36.3 #1)
<4>[  211.060307] PC is at nvhost_read_module_regs+0x3c/0x70
<4>[  211.066104] LR is at __mutex_unlock_slowpath+0xd8/0x15c
<4>[  211.071685] pc : [<c01d391c>]    lr : [<c0442b68>]    psr: 20000013
<4>[  211.071705] sp : d2421da8  ip : 00000000  fp : d2421dd0
```

Fig. 1. The logs showing the vulnerability 1's position

```
if ((fd = open (TARGET, O_RDONLY))< 0)
   { perror (``cannot open it \n");
     ret = ioctl (fd, NVHOST_IOCTL_CTRL_MODULE_REGRDWR, &cmd);
     memcpy(newvalues, oldvalues, 0x10);
     dumpbin(oldvalues, 0x10);
     cmd.id = 0;  cmd.numoffsets = 1;  cmd.blocksize=0x10; cmd.offsets = &offset;
     cmd.values = &newvalues;    cmd.write = 1;    newvalues[0] = 0;
    ret = ioctl (fd, NVHOST_IOCTL_CTRL_MODULE_REGRDWR, &cmd);
     dumpbin(newvalues, 0x10);
    setuid(0);
    system("/system/bin/sh"); }
```

Fig. 2. The main exploit code of regrdwr.c

IV. Exploiting the Vulnerabilities

A. Exploiting Vulnerability #1

Since Android is based on a modified Linux kernel and thus it applies the Discretionary Access Control (DAC) on the filesystem level, which is based on user IDs (uid) and group IDs (gid). If the uid = 0, this means that the user get root-level privilege, which is the goal of exploiting privilege escalation vulnerabilities.

The first vulnerability is referred to as the privilege escalation vulnerability. Since we know this vulnerability related to the address, we can scan the kallsyms log and find the offset of the sys-setuid function. This means that we can find out the address of the sys-setuid function. If we insert malicious code here, then we can execute the malicious code to change the uid. We overwrite the code of setuid using newvalues[0] = 0 to get the root privilege, then setuid = 0. After that we create a shell. Now, we have full control of the Android OS. We verified the vulnerability using a fixed offset 0x0800000. Fig. 1 show the logs, which tell us the position of the vulnerability in line 9 and it is a buffer overflow vulnerability. We exploited the first vulnerability in regrdwr.c. Fig. 2 show the main code of regrdwr.c.

By exploiting code as described above, we confirmed the privilege escalation vulnerability on several currently available versions of Android OS. We conducted the tests by using a real Android device – a Samsung Galaxy tablet 10.1.

Fig. 3 is a screen copy of the result of exploiting the vulnerability. Fig. 3 shows that after running the exploit code, the uid was changed from 7d0 to 0. This validated that we successfully escalated to root privileges.

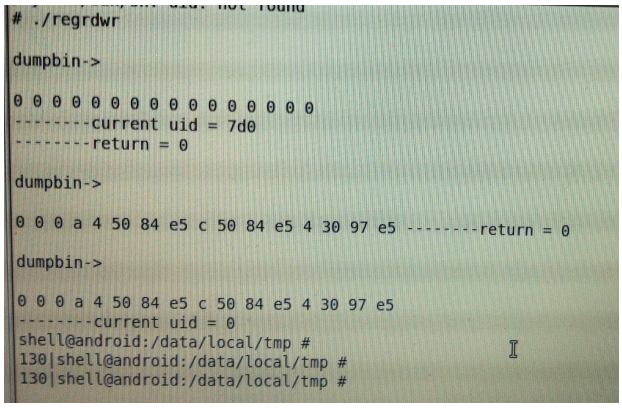

Fig. 3. The changing UID attack

B. Exploiting Vulnerability #2

The second vulnerability is referred to as the DoS vulnerability. We can easily exploit this vulnerability by a simply fuzzy test. We tested the second vulnerability with nvfuzz.c. Fig.4 shows the main code of nvfuzz.c. The Android OS crashed several times. Fig.5 shows the kernel logs. If a hacker inserts malicious codes in Android applications that are available. to all the users, then thousands of Android devices with Nvidia Tegra chips will crash.

If a hacker combines these two vulnerabilities, then he can crash a lot of devices, disable anti-virus software, install any malware, create malware, and even publish malicious applications in Android application market with paying $25 register fee.

We confirmed the DoS vulnerability on several currently available versions of Android Honeycomb OS by fuzzy tests. We run the tests by using a real Android device – a Samsung Galaxy tablet 10.1.

Fig. 5 shows that after we run the exploit code, the kernel is panic and the system is reset. If we continually run the exploit code, the system cannot work any more. Hence, this is an exploit that leads to the DoS attack.

We tested on different versions of Android Honeycomb OS. For each version, we run the test many times. All our tests have caused Android Honeycomb to crash. After we explored the drivers source code for Tegra, we believe the two vulnerabilities are universal in Android device with Tegra.

V. Countermeasures

In this Section, we describe two approaches to fix the two vulnerabilities described in Section IV. The fix for the privilege escalation vulnerability consists of checking whether offs in function nvhost_read_module_regs and function nvhost_write_module_regs is out of the boundary. The fix for DoS vulnerability is to restrict the size of _IOC_SIZE(cmd).

A. Fix for the Privilege Escalation Vulnerability

As mentioned in Section IV, the nvhost_read_module_regs and nvhost_write_module_regs functions do not perform any

```
for (i1 = 0; i1 < 4; i1++)
{       for (i2 = 0; i2 < 2; i2++)
        {   for (i3 = 0x30; i3 < 0x7a; i3++)
            {  l4stop = 0x100;
               for (i4 = 0; i4 < l4stop; i4++)
               { icode = i1s[i1] * 0x1000000 + i2s[i2] * 0x10000 + i3 * 0x100 +i4;
               printf ("----------------------------------------start to fuzzing(%s)
               ioctl ( %x, %x) \n", files [ fileindex], fd, icode);
               for (i = 0; i < 8; i++)
                  {  ret = ioctl (fd, icode, data[i]);
                     if (ret != 0 && ret != -1)
                     {       l4stop = 0x100;
                             if (debug)
                             perror ("error  ioctl \n");  continue;  }
                     else if (ret == -1)
                  {  continue;  }
                     else {  l4stop = 0x100;  } } } } } }
```

Fig. 4. The main code of nvfuzz.c

```
[<c0008b2c>] (start_kernel+0x294/0x2ec) from [<00008080>] (0x8080)
[DBG] upload cause : UPLOAD_CAUSE_KERNEL_PANIC
(kernel_sec_set_upload_cause) : upload_cause set c8
Rebooting in 10 seconds..misc_sec_operation d36124e0 1ff000 52 0
misc_sec_operation: filp_open failed. (-13)
sec_open_param PARAM OPEN FAIL
(kernel_sec_get_debug_level) The debug value is invalid(0x0)!! Set default level(LOW)
(kernel_sec_hw_reset) Upload Magic Code is cleared for silet reset.
(kernel_sec_hw_reset)
(kernel_sec_hw_reset) The forced reset was called. The system will be reset !!
```

Fig. 5. Logs for Experiment 2 - Dos Vulnerability

specific check on the variable offs. Hence, the fix is to add a check of the variable offs and see if it is out of the boundary.

B. DoS vulnerability fix

Because the program fails to check the size of _IOC_SIZE(cmd), a malicious user can send a very long cmd to overflow the kernel. This will cause a kernel panic. If we add length check to check the size of _IOC_SIZE(cmd), then this problem is solved.

C. Testing the Countermeasures

We implement the two countermeasures in our Samsung Galaxy tablet 10.1. In particular, for the Android Honeycomb 3.0.1 version, we build one patched version that includes two patches, by recompiling Android from scratch. Our tests show that both patches are effective and prevent the exploit codes, thereby fixing the two vulnerabilities.

VI. Related Works

Security of Android platform has been studied by many researchers. There are three main trends:

- static analysis
- security scheme assessment
- malware and virus detection

Static analysis includes using the white box or black box methodologies to detect malicious behaviors in Android applications before installing them on the devices. One paper by Enck et al. [4] have a horizontal study of Android applications to discover stealing of personal data. Fuchs et al. [5] propose Scandroid, which is an automatical reasoning tool to find security violations of Android applications. Static analysis also could help identifying vulnerabilities in the kernel.

The second trend of current research is to study access control and permission schemes of Android. For example, [6] proposes a scheme to assess the actual privileges of Android applications and develops Stowaway, which is a tool to detect over-privilege in compiled Android applications. Nauman et al. [7] propose Apex, which is a policy enforcement framework for Android that allows a user to fine-grained grant permissions to applications and impose constraints on the usage of resources. Ongtang et al. [8] present an infrastructure named Secure Application INTeraction (SAINT) to govern permission assignment during installation. Many works focus on the privilege escalation issues. Bugiel et al. [9] propose a security framework named eXtended Monitoring on Android (XManDroid) to extend the native monitoring mechanism of Android for detecting the privilege escalation attack. The privilege escalation attack on Android was first proposed by Davi et al. [10] in which they demonstrated an example of the attack. They showed that a genuine application exploited at runtime or a malicious application can escalate granted permissions. However, they did not suggest any defense for the attack in the paper. Underprivileged applications under the malicious user's control can perform operations indirectly by invoking other applications possessing desired privileges. The attacks published are unauthorized phone calls [31], text message sending [8] to illegally downloading the malicious files [34], and context-aware voice recording [36], [32]. Most privilege escalation attacks exploit vulnerable interfaces of privileged applications. This attack is often referred to as confused deputy attack [33], [30]. However, in general, the adversary can design his own malicious applications which collaborate to mount a collusion attack [36]: applications with uncritical permissions can collude to generate a joint set of permissions that enables them to perform unauthorized actions. Some collusion applications [32] may exploit covert channels of the Android core system to avoid detection.

Note that the Android application distribution model allows anyone who has registered as an Android developer (and paid $25 fee) to publish applications on the Android market. This scheme allows adversaries to easily upload malicious applications on the market store: For instance, the recent Android DroidDream Trojan (containing a root exploit) has been identified in over 50 official Android market applications and has been downloaded more than 10,000 times before it has been detected [29]. Literature [11] focuses on possible threats and solutions to mitigate privilege escalation problem proposed by literature [12].

In the third trend - virus and malware detection, Dagon et al. [13] assess many mobile viruses and malware that could potentially affect Android devices. Crowdroid [14] is proposed as a malware detector executing a dynamic analysis on application behaviors. Schmidt et al. [15] inspect Android executables to extract their function calls and compare them with malware executables for classification purpose. Specific malware signatures for exploiting the vulnerabilities described

in this paper could be generated too. All these works are to protect user's privacy and security. The authors of paper [16] think that we may need an **original** privacy mode in Android smartphones. Literature [23] presents a DoS attack that makes devices totally unresponsive by repeatedly forking the Zygote process. The vulnerabilities disclosed in this paper require that the USB development debugging function is enabled. For devices without Android Developer Bridge enabled, malicious users still can use the recovery boot method [26] to exploit the two vulnerabilities.

VII. Conclusions

Android operation system is widely used in smartphones and tablet devices. In this paper, we presented two new vulnerabilities in Tegra driver programs located in Android kernel. The first vulnerability can be used to escalate the kernel privileges. The second vulnerability can be used to launch the deny of service (DoS) attack. To verify these vulnerabilities, we successfully exploited the two vulnerabilities on several versions of Android by using a real device - a Galaxy tablet device. We reported the two vulnerabilities to the Android security team. Furthermore, we provided security patches to fix the two vulnerabilities and we confirmed that the patches work.

Acknowledgment

This work was supported in part by the US National Science Foundation under grants CNS-0963578, CNS-1022552, CNS-1065444, CNS-1239108, IIS-1231680 and CNS-1218718.

References

[1] http://soltesza.wordpress.com/2010/01/08/nvidia-leading-the-smartbook-revolution/

[2] IDC. Worldwide Smartphone 2012-2016 Forecast and Analysis. http://www.idc.com/getdoc.jsp?containerId=233553

[3] Gartner Group. Press Release, November 2011. Available at http://www.gartner.com/it/page.jsp?id=1848514.

[4] W. Enck, D. Octeau, and P. McDaniel et al. A study of android application security. In Proc. of the 20th USENIX conf. on Security, pp. 21-21, Berkeley, CA, USA, 2011. USENIX Association.

[5] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications.

[6] A. P. Felt, E. Chin and S. Hanna et al. Android permissions demystified. In Proc. of the 18th ACM conf. on Computer and communications security, pp. 627-638, 2011.

[7] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In Proc. of the 5th ACM Symp. on Information, Computer and Communications Security, pp. 328-332, New York, NY, USA, 2010. ACM.

[8] M. Ongtang, S. Mclaughlin, and W. Enck et al. Semantically rich application-centric security in android. In Annual Computer Security Applications Conference, 2009.

[9] S. Bugiel, L. Davi, and A. Dmitrienko et al. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt, Apr 2011.

[10] L. Davi, A. Dmitrienko, and A. Sadeghi et al. Privilege escalation attacks on android. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana Ilic, editors, Information Security, vol. 6531 of LNCS, pp. 346-360. 2011.

[11] E. Chin, A. P. Felt, and K. Greenwood et al. Analyzing inter-application communication in Android. In Proc. of the 9th Intl. Conf. on Mobile systems, applications, and services, pp. 239-252, New York, NY, USA, 2011. ACM.

[12] A. Shabtai, Y. Fledel, and U. Kanonov et al. Google android: A state-of-the-art review of security mechanisms. CoRR, abs/0912.5101, 2009.

[13] D. Dagon, T. Martin, and T. Starner. Mobile phones as computing devices: The viruses are coming! IEEE Pervasive Computing, vol. 3, no. 4, pp. 11-15, 2004.

[14] I. Burguera, U. Zurutuza, and S. Nadjm-Therani. Crowdroid: behaviorbased malware detection system for android. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, 2011.

[15] A.-D. Schmidt, R. Bye, and H.-G. Schmidt et al. Static analysis of executables for collaborative malware detection on android. In Communications, 2009. IEEE Intl. Conf. on Communications, pp. 1-5, June 2009.

[16] Y. Zhou, X. Zhang, and X. Jiang et al. Taming information-stealing smartphone applications (on android). In Proc. of the 4th Intl. Conf. on Trust and trustworthy computing, pp. 93-107, 2011.

[17] Apple App Store. http://www.apple.com/iphone/ apps-for-iphone/.

[18] M. Egele, C. Kruegel, and E. Kirda et al. PiOS: Detecting Privacy Leaks in iOS Applications. In Proc. of the 18th Annual Network and Distributed System Security Symp., February 2011.

[19] W. Enck, P. Gilbert, and B.-G. Chun et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In Proc. of the 9th USENIX Symp. on the USENIX Symp. on Operating Systems Design and Implementation. Vancouver, BC, Oct. 2010.

[20] K. Mahaffey and J. Hering. App Attack-Surviving the Explosive Growth of Mobile Apps. https://media.blackhat.com/bh-us-10/presentations/Mahaffey_Hering/Blackhat-USA-2010- Mahaffey-Hering-Lookout-App-Genomeslides.pdf.

[21] Y. Zhou, X. Zhang, and X. Jiang et al. Taming Information-Stealing Smartphone Applications (on Android). In Proc. of the 4th Intl. Conf. on Trust and Trustworthy Computing, June 2011.

[22] IPhone Stored Location in Test Even if Disabled. http://online.wsj.com/article/SB10001424052748704123204576283580249161342.html.

[23] A. Armando, A. Merlo, and M. Migliardi et al. Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures). In IFIP SEC 2012 27th International Information Security and Privacy Conf., D. Gritzalis, S. Furnell, and M. Theoharidou (Eds.), pp. 13-24, June 2012, Heraklion, Greece, IFIP Advances in Information and Communication Technology, Vol. 376, Springer, 2012.

[24] http://developer.android.com/guide/topics/security/security.html

[25] Rooting the droid without rsdlite. http: //androidforums.com/droid-all-thingsroot/ 171056-rooting-droid-withoutrsd- lite-up-including-frg83d.html, Dec. 2010.

[26] T. Vidas, C. Zhang, and N. Christin. Towards a general collection methodology for android devices. DFRWS 2011, Aug. 2011.

[27] A. Waqas. Root any android device and samsung captivate with super one-click app. http://www.addictivetips.com/mobile/ root-any-android-device-and-samsungcaptivate- with-super-one-click-app/, Oct. 2010.

[28] https://opensource.samsung.com/reception/receptionSub.do?method =list&menu_item=mobile&classification1= % mobile_phone &classification2=&classification3

[29] T. Bradley. Droiddream becomes android market nightmare. http://www.pcworld.com/ businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html, 2011.

[30] M. Dietz, S. Shekhar, and Y. Pisetsky et al. Quire: Lightweight provenance for smartphone operating systems. In 20th USENIX Security Symp., 2011.

[31] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State Univ., Sep 2008.

[32] G. Halfacree. Android trojan captures credit card details. http://www.thinq.co.uk/2011/1/20/android- trojan-captures-credit-card-details/, 2011.

[33] N. Hardy. The confused deputy: (or why capabilities might have been invented). SIGOPS Oper. Syst. Rev., 22:36-38, Oct. 1988.

[34] A. Lineberry, D. L. Richardson, and T. Wyatt. These aren't the permissions you're looking for. BlackHat USA 2010. http://dtors.files.wordpress.com/ 2010/08/blackhat-2010-slides.pdf, 2010.

[35] Nils. Building Android sandcastles in Android's sandbox. BlackHat 2010. https://media.blackhat.com/bh-ad-10/Nils/Black- Hat-AD-2010-android-sandcastle-wp.pdf, 2010.

[36] R. Schlegel, K. Zhang, and X. Zhou et al. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In Proc. of the 18th Annual Network and Distributed System Security Symp, pp. 17-33, Feb. 2011.