Smartphone Malware Evolution Revisited: Android Next Target?

Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Leonid Batyuk,
Jan Hendrik Clausen, Seyit Ahmet Camtepe, and Sahin Albayrak
Technische Universitä Berlin - DAI-Labor
Ernst-Reuter-Platz 7, 10587 Berlin
{aubrey.schmidt, hans-gunther.schmidt, leonid.batyuk,
jan.clausen, ahmet.camtepe, sahin.albayrak}@dai-labor.de

Can Yildizli
Sabanci University, Istanbul,
canyildizli@su.sabanciuniv.edu

Abstract

Smartphones started being targets for malware in June 2004 while malware count increased steadily until the introduction of a mandatory application signing mechanism for Symbian OS in 2006. From this point on, only few news could be read on this topic. Even despite of new emerging smartphone platforms, e.g. Android and iPhone, malware writers seemed to lose interest in writing malware for smartphones giving users an inappropriate feeling of safety.

In this paper, we revisit smartphone malware evolution for completing the appearance list until end of 2008. For contributing to smartphone malware research, we continue this list by adding descriptions on possible techniques for creating the first malware(s) for Android platform[1]*. Our approach involves usage of undocumented Android functions enabling us to execute native Linux application even on retail Android devices. This can be exploited to create malicious Linux applications and daemons using various methods to attack a device. In this manner, we also show that it is possible to bypass the Android permission system by using native Linux applications.*

1 Introduction

Smartphones get increasingly popular which also attracted malware writers beginning from June 2004. From this point on, malware count increased steadily while main target remained Symbian OS[2]. After the introduction of application signing, the amount of new appearing malware decreased while only few news can be read up to today. Even since new emerging platforms seem to be a valuable target, e.g. Android and iPhone, only first vulnerabilities were published but no fully working malwares.

The contribution of this paper is twofold. We start with revisiting the smartphone evolution starting in June 2004 and enter this evolution by presenting the first running malware for Android. Our intention is of course not to start a new wave of emerging malwares for Android, instead, we want to contribute to current smartphone malware research by pointing to possible vulnerabilities concerning the Android platform. This is of special interest since this new platform targets for fully integrating Internet and general network services to its devices. Attacking this ability and also the increasing user dependence on these devices can result in significant denial of service as well as high costs at user side due to malicious service usage.

This work is structured as follows. In Section 2, we present related smartphone malware research. In Section 3, we revisit smartphone malware history for showing what happened up to date in this field. In Section 4, we give a brief introduction to the Android platform followed by Android-specific software engineering aspects that are needed in order to create malware. In Section 5, we describe how to exploit the aforementioned techniques in order to create Android malware. In Section 6, we conclude.

2 Related Work

Since our work lies in the field of smartphone malware, several related papers can be identified. Starting with the first wave of Symbian OS malwares, several authors pointed to the "new" threat targeting smartphones, e.g. Dagon et

[1] http://code.google.com/android/
[2] http://www.symbian.org/index.php

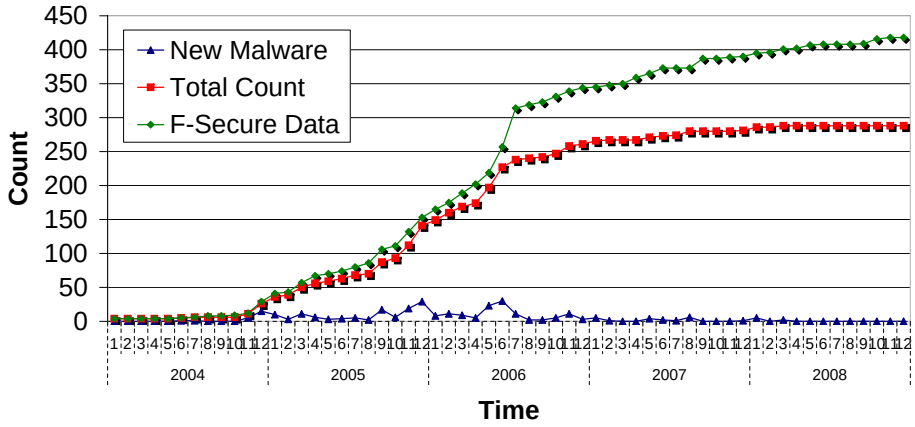

Figure 1: Mobile malware evolution basing on published malware including descriptions on their behavior. F-Secure data was added for comparison.

al. [7], Jamaluddin et al. [12], Piercy [21], Niemelä [20], Leavitt [14], and Hypponen [11].

Overviews on smartphone malware appearance were given by Töyssy et al. [26], Gostev [9, 10], Fleizach et al. [8], Lawton [13], Schmidt et al. [23], and Shih et al. [24] while most of them end in 2005 or 2006. In this work, we update these overviews by extending the list of appearances to the end of 2008 while practically adding a new entry for the beginning of 2009.

Android, but also iPhone malware propagation is still a basically not investigated field of research. Since both platforms mainly use a online store for distributing software, metrics have to be found for predicting or simulating malware propagation. Valuable input is given by Mickens et al. [15], Bulygin [6], and Wang et al. [27], who give interesting insights into propagation models and estimations.

The possibility of attacking smartphones was investigated by several researchers where especially the continuous work of Mulliner et al. 2006 [16, 19, 17, 18] has to be noted since essential work concerning Windows Mobile and Symbian OS was presented. Related work was also published by Racic et al. [22] who used MMS in order to deplete the battery of mobile phones. Becher et al. [5] presented a promising approach for creating a worm for Windows Mobile. Unfortunately, they were lacking the appropriate exploit for making a fully working malware. Jesse D'Aguanno[3] gives detailed information on how to attack RIM Blackberry supporting networks[4].

[3]presented as speaker with the pseudonym "x30n"

[4]proof-of-concept at `http://www.praetoriang.net/presentations/blackjack.html`

3 Revisiting Smartphone Malware Evolution

Initial work on showing smartphone malware evolution was published by Gostev [9] from Kaspersky Lab. Key results for the time span from June 2004 until August 2006 were that smartphone got an increasingly popular target for attackers where mainly Symbian OS malwares appeared. Our paper extends this work from a perspective not having the same access to malware databases as most anti-virus product vendors have. We noticed a great discrepancy between published malwares and corresponding available descriptions on their behaviors. Especially in the last two years, descriptions on the behaviors got scarce without obvious reason.

For statistical purpose, we gathered all published malware *descriptions*[5] from various web pages, e.g. from F-Secure, Kaspersky, McAfee, Symantec, Sophos, and similar, for identifying key aspects of mobile malware. One obvious aspect is their appearance in time. Figure 1 shows mobile malware evolution from January 2004 to December 2008 based on published mobile malware with available behavior description. We found 288 smartphone malwares until end of 2008 where peeks in new appearing malware can be found at the end of 2005 and in the middle of 2006. It is probable that these peeks were caused by the introduction of a certificate-based signing of application for Symbian OS[6] where malware writers might have feared a decreasing number of possible victims. In the signing process, a trusted Symbian partner checks the complete source code and binaries for meeting certain criteria, like being free from memory leaks and abusive methods. If the check is successful, the application gets signed with an certificate

[5]malwares lacking descriptions were ignored

[6]`http://www.symbiansigned.com`

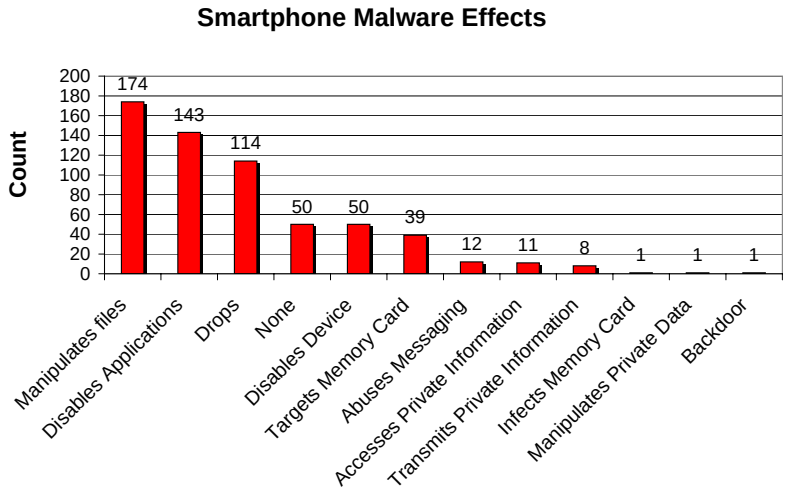

Figure 2: Smartphone malware effects

and stays clearly identifiable through a given unique ID. Additionally, signing restricts access to sensitive function calls from certain APIs, e.g. network control, preventing abusive usage of these. Application signing gets mandatory for the current Symbian S60 *3rd* Version which is installed on most Nokia smartphones since the end of 2005.

For comparison, we requested the corresponding numbers from F-Secure Research in Helsinki[7]. Comparing the numbers from Figure 1, you can see that F-Secure counted 418 malwares, 130 more than we found, showing that there are several malwares without publicly available descriptions. Additionally, following the F-Secure numbers, in the middle of 2006 more than 100 new malwares appeared.

Based on published malware descriptions, we listed the malware effects which can be seen on Figure 2. Please note that the categories are not disjunct, therefore the count of malware having certain effects exceeds our total count of 288 malwares. Several different malicious behaviors were recognized while more than half of the malwares manipulated files for achieving application or device disabling. Another interesting point is that 50 malwares did not have a malicious behavior except their propagating functionality.

Smartphone malware uses various channels for infecting new devices. What most malwares, especially for Symbian OS, have in common is that they require an installation file for propagation. Additionally, Bluetooth and MMS were used for propagating these malwares which can be seen on Figure 3.

All malwares basing on (Symbian OS) installation files explicitly need user interaction for installing on the system. Therefore, most smartphone malwares are categorized as "Trojan horses" (84%), see Figure 4. Even worms (15%)

[7]We want to kindly thank Jarno Niemela for providing us these information

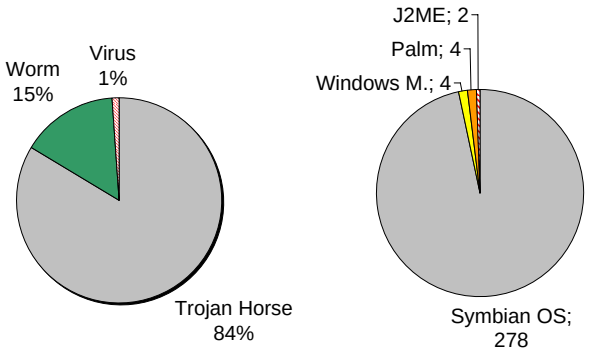

Figure 4: Left: appeared malware categories, right: malware per platform

need user interactions in order to get installed. Hence, propagation schemes cannot be compared with Windows worms using system vulnerabilities.

Interestingly, most of the malwares target Symbian OS (283 malwares) where only 4 Windows Mobile and 2 Java ME malwares were recognized. The payload of the Windows Mobile and Java ME malwares included remote access, file deletion, and abuse of the SMS in order to charge high service usage rates.

Coming back to Figure 1, malware appearance decreased starting from the middle of the year 2006. Until end of 2008, only about 100 new malwares appeared while between the same time span from end of 2004 to middle of 2006 about 300 emerged. The reason for this can be seen in the certification system of Symbian OS 3rd where devices running this operating system version gained more and more market share at this time. Since this OS version is not vulnerable to the former malware and less possible vic-

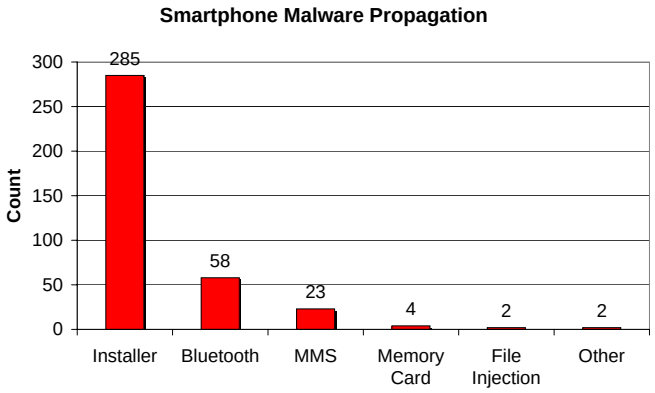

Figure 3: Malware propagation

tims were targetable, this platform seemed less attractive. Only the news that a spyware got certified for Symbian OS 3rd, called FlexiSpy [25], recalled the existing threat to this new age of smartphone OS.

Besides the recognized malware "in the wild" several research activities aimed for bypassing the security systems of smartphones. One of the latest work of Collin Mulliner resulted in the ability to bypass the security mechanisms of Symbian OS 3rd which he presented on Black Hat Conference 2008 in Japan[8]. Short after this, in February 2009, the first malware targeting Symbian OS 3rd appeared using a valid certificate[9].

4 Software Engineering Aspects of Android

For continuing the evolution of smartphone malware by adding a new target platform we need to discuss essential characteristics of Android OS which we already did in [4]. Therefore, we present an excerpt from [4] in this section where we focus on discussing possible ways to develop native Linux software for Android. This will be exploited in the following section in order create malware.

4.1 Android Linux Application Development

From a Linux perspective, Android provides a complete operating system running on an ARM-Architecture. Compiling software for ARM requires a specific environment. In following sections, we will describe a possible way for compiling software successfully within an ARM-compatible environment. Additionally, we present relevant information on the operating system level and describe how to bridge Android Java applications to Android native Linux applications.

4.1.1 Base Environment

Ubuntu i686 GNU/Linux [3], a Linux-distribution provided and supported by Canonical, provides the basis for all further steps. Based on the Intel-architecture, a vast amount of tools, especially for creating and compiling software, can be obtained through Ubuntu's package repositories.

4.1.2 Scratchbox Cross-Compilation Toolkit

Scratchbox [2] not only provides the necessary compilers and linkers, it also provides a complete environment simulating an ARM platform-based operating system. All tools compiled within this environment can be tested immediately giving a very fast feedback to the developer. Once the Ubuntu package repository has been extended by the official Scratchbox repository, all necessary files for Scratchbox can be easily installed via Ubuntu's package management tools. Scratchbox offers a wide variety of possible compilers, in different versions and characteristics. After installation, a user account has to be created for use within the Scratchbox environment. Shortly after logging into the new environment, preliminary steps are required: select the desired compiler and add additional tools if wanted (`strace`, `gdb`). From this point, source code can be compiled as usual, no specific parameters have to be provided. The host and build type are distinguished automatically, standard locations for installing binaries, libraries, etc. are provided. As long as the given source code is ARM-compatible, it will most likely compile within Scratchbox without any significant problems. Having successfully compiled all files, these can be packed into an archive for being transferred to the Android environment for deployment.

[8] `http://mulliner.org/symbian/feed/CollinMulliner_Exploiting_Symbian_BlackHat_Japan_2008.pdf`

[9] `http://www.f-secure.com/weblog/archives/00001609.html`

4.1.3 Important Facts for Native Development on Android

Filesystem Specifics

Google provides an ARM Linux with a filesystem layout which greatly differs from usual Linux filesystem layouts:

- System relevant files are found in the System image, mounted to `/system` (binaries are, for the most part, found in `/system/bin`, libraries reside in `/system/lib`, configuration files in `/system/etc`, etc.)
- User data relevant files and applications reside within the user data image, mounted to `/data`.

Handling these changes does not require much adaptation.

4.1.4 Bridging Between Java and Linux

For creating a Android malware that is distributed via the corresponding online store[10], Java to `C` communication might get necessary, at least to start the malware. Therefore, we present two possible solutions - Java Native Interface (JNI), a commonly accepted standard in the Java development community, and named pipes which are commonly used on unixoid systems for simple inter-process communication.

Java Native Interface (JNI)

JNI is used to call native functions of the underlying operating system. Using this interface, the developer risks losing the platform independence of Java unless the native call exists on all intended platforms. At the moment, JNI is not supported on Android although it is used across the system. Following Romain Guy, an Android developer at Google, Android currently uses JNI only for the framework and not for the applications [1]. Nonetheless, Google seems to be working on a *native* SDK officially providing JNI calls.

Despite the official Google statement that JNI is currently not supported for user applications and won't work, we have successfully compiled a Java application which uses a custom JNI shared library. It was possible to install and run the application on the retail G1 without any further modification. The native component has been packaged into an APK as a raw binary resource and unpacked upon first execution of the Java program. After doing so, it is possible to load the shared object as a JNI library via invoking `java.lang.System.load(String filename)`.

From our point of view, JNI is rather hard to implement, since compiling a shared library for Android is a challenging task because of the unusual page alignment. But, it is most probably going to become the only official way to include native code in Android applications, and also has shown good performance.

[10] Android Market, http://market.android.com

Pipes

Using pipes is a commonly used technique in Linux and Unix systems to allow communication between separated processes.

For creating named pipes the command `mkfifo` can be used. Additional information on pipes can be found in the corresponding man pages. Using pipes on Android for communication between Java and native executables is rather straight-forward, since Java provides a lot of convenient writer, reader, and stream classes. But, when it comes to deploying and executing the binary in the restricted environment where an application can only write to its own folder, this approach requires decent programming skills. The pipe technique is much more complex in its deployment than JNI, but it gives the developer a possibility to start a persistent daemon on a Linux layer while being independent from the Dalvik application lifecycle.

5 Creating Android Malware

We exploit the aforementioned techniques in order to create a "social engineering"-based malware. The concept of this malware bases on the use of a malicious Linux application packed into the installation file of a valid Android application. The payload of this malware can be various, where examples are shown next on.

First of all, a hosting application is needed that is created as standard Android Java application. This application can consist of a game or tool containing basically only few lines of code that are important for getting the malware run. The malicious Linux binary itself is packed as "raw resource" into this Java application, e.g as .png file, which can be seen on Figure 5. After installation, the Java application has to be executed once in order to rename the resource file into the appropriate binary. After renaming the file, the file has to be made executable which is currently impossible from within Java. This is where the sources of Android OS are needed - using the Android Build System instead of Ant, it is possible to utilize the class `android.os.Exec`, which is not included in the SDK classpath. It allows execution of native binaries as a subprocess of the Java application. Using `Exec.createSubprocess("/system/bin/sh", ...)`, it is possible to start the Linux shell and utilize `chmod` to make the binary executable. Afterwards, the application can be launched on a retail G1 Android smartphone.

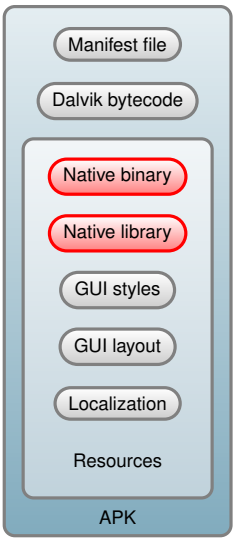

Figure 5: The Android Installation file containing malicious components

Investigating the possible payload of the Linux application, we found several suitable examples that can be applied. The first one bypasses the Android permission system. This permission system basically enforces restrictions on certain Java calls. The restrictions are set in the `AndroidManifest.xml` file of the application.

It is possible to read several files, e.g. */proc/* and */sys/*, from a Linux-level application. This can lead to the following bypass example: while not having the *android.permission.BATTERY_STATS* permission you can still read the battery status from */sys/class/power_supply/*. This is not a critical issue but further investigation might point to more concerning bypasses.

Another possible payload bases on the fact that the G1 and similar devices basing on the same ARM architecture lack the Floating Point Unit (FPU) in order to compute floating-point operations. Therefore, it is possible to write a battery depleting routine basing on this. Our latest results show that combining graphical applications using FPU operations is a fast way to deplete the battery, but since graphical applications rely on Android Java API, currently, this cannot be taken into account from a Linux perspective.

Our last example uses a *rooted*[11] G1 Android Smartphone. It is possible to execute arbitrary ARM instructions on the G1 device, as well as C programs. As an example, we were able to identify shellcode for rebooting a G1 device. Currently, we are still not able to do this on a retail phone where our work is still in progress. The shellcode to reboot a G1 can be seen in Listing 1 and can be easily executed from within Android Linux applications. Additionally, we found out that Linux applications can be started as daemon, rendering the rooted device useless when applying the reboot code.

Listing 1: Reboot shell code for rooted G1

```
1  char code[] =
2  "\x01\x10\xa0\xe1" //nopslide begin
3  "\x01\x10\xa0\xe1"
4  "\x01\x10\xa0\xe1"
5  "\x01\x10\xa0\xe1"
6  "\x01\x10\xa0\xe1"
7  "\x01\x10\xa0\xe1"
8  "\x01\x10\xa0\xe1" //nopslide end
9  "\x90\x00\x2d\xe9" //save r4 r7
10 "\x58\x70\xa0\xe3" //set r7 to 58
11 "\x10\x00\x9f\xe5" //set r0 to arg1
12 "\x10\x10\x9f\xe5" //set r1 to arg2
13 "\x10\x20\x9f\xe5" //set r2 to arg3
14 "\x10\x30\x9f\xe5" //set r3 to arg4
15 "\x00\x00\x00\xef" //svc 0
16 "\x0e\xf0\xa0\xe1" //retn
17 "\xad\xde\xe1\xfe" //arg1
18 "\x69\x19\x12\x28" //arg2
19 "\xdc\xfe\x21\x43" //arg3
20 "\x00\x00\x00\x00" //arg4
21 ;
```

5.1 Discussion on Possible Countermeasures

Countermeasuring the concept of using JNI in Android is not trivial. JNI is a fundamental part of the Android Framework being included in many system applications and services. So removing this functionality cannot be considered. As an alternative, a slightly complex approach would be to pre-check Android applications before publishing them in the corresponding online store. These checks could be performed as static or dynamic analysis with the aim of identifying usage of undocumented API calls. Positive detection might result in an application or developer[12] ban. Another approach might be to restrict execution of resource files of user applications.

6 Conclusion and Future Work

In this work we started with updating smartphone malware evolution to the end of 2008. This evolution is continued by us by adding a new platform affected by smartphone

[11] A special operating system image is installed on the device providing root privileges to the user

[12] Developers are identifiable through signatures which are used to sign applications

malware, namely Android. By using undocumented Android Java functions, we created a malware that can show various malicious behaviours. We also show that currently, it is possible to bypass the Android permission system by using native applications, where we have to note that our current findings are not critical.

Since we found a way to reboot rooted Android phones via shellcode, our future work will focus on transferring the same functionality to a retail Android phone. Since this shellcode can "brick"[13] an Android phone, it is really important to learn, whether retail phones are also threatened by this and how this could be prevented.

References

[1] Android developer mailing list post. http://groups.google.com/group/android-developers/browse_thread/thread/f87e6fce2b26db36.

[2] Scratchbox. http://www.scratchbox.org/.

[3] Ubuntu home page. http://www.ubuntu.com/.

[4] L. Batyuk, A.-D. Schmidt, H.-G. Schmidt, A. Camtepe, and S. Albayrak. Developing and benchmarking native linux applications on android. In *MobileWireless Middleware, Operating Systems, and Applications*, 2009.

[5] M. Becher, F. Freiling, and B. Leider. On the effort to create smartphone worms in windows mobile. In *Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC*, pages 199–206, 20-22 June 2007.

[6] Y. Bulygin. Epidemics of mobile worms. In *Proceedings of the 26th IEEE International Performance Computing and Communications Conference, IPCCC 2007, April 11-13, 2007, New Orleans, Louisiana, USA*, pages 475–478. IEEE Computer Society, 2007.

[7] D. Dagon, T. Martin, and T. Starner. Mobile phones as computing devices: The viruses are coming! *IEEE Pervasive Computing*, 3(4):11–15, 2004.

[8] C. Fleizach, M. Liljienstam, P. Johansson, G. M. Voelker, and A. Mehes. Can you infect me now? malware propagation in mobile phone networks. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM 2007)*, Alexandira, Virginia, USA, nov 2007. ACM.

[9] A. Gostev. Mobile Malware Evolution: An Overview, Part 1, Sept. 2006.

[10] A. Gostev. Mobile Malware Evolution: An Overview, Part 2, Oct. 2006.

[11] M. Hypponen. Malware goes mobile. *Scientific American*, November 2006:70–77, November 2006.

[12] J. Jamaluddin, N. Zotou, R. Edwards, and P. Coulton. Mobile Phone Vulnerabilities: A New Generation of Malware. In *Proceedings of the 2004 IEEE International Symposium on Consumer Electronics*, pages 199–202, Sept. 2004.

[13] G. Lawton. Is it finally time to worry about mobile malware? *Computer*, 41(5):12–14, 2008.

[14] N. Leavitt. Mobile phones: The next frontier for hackers? *IEEE Computer*, 38(4):20–23, 2005.

[15] J. W. Mickens and B. D. Noble. Modeling epidemic spreading in mobile environments. In *WiSe '05: Proceedings of the 4th ACM workshop on Wireless security*, pages 77–86, New York, NY, USA, 2005. ACM Press.

[16] C. Mulliner. Exploiting pocketpc, 2005. Talk on WhatTheHack 2005, http://wiki.whatthehack.org/images/c/c0/Collinmulliner_wth2005_exploiting_pocketpc.pdf.

[17] C. Mulliner. Advanced attacks against pocketpc phones. 2006.

[18] C. Mulliner. Exploiting symbian: Symbian exploitation and shellcode development, 2008. Talk on BlackHat Japan 2008, http://mulliner.org/symbian/feed/CollinMulliner_Exploiting_Symbian_BlackHat_Japan_2008.pdf.

[19] C. Mulliner and G. Vigna. Vulnerability Analysis of MMS User Agents. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 77–88, Washington, DC, USA, 2006. IEEE Computer Society.

[20] J. Niemelä. What makes symbian malware tick. In *Proceedings of the 15th Virus Bulletin Conference*, pages 314–322. Virus Bulletin Ltd., 2005.

[21] M. Piercy. Embedded devices next on the virus target list. *IEE Electronics Systems and Software*, 2:42–43, Dec.-Jan. 2004.

[22] R. Racic, D. Ma, and H. Chen. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery. In *Proceedings of the Second IEEE Communications Society / CreateNet International Conference on Security and Privacy in Communication Networks (SecureComm), Baltimore, MD*, Aug. 2006.

[23] A. Schmidt and S. Albayrak. Malicious software for smartphones. Technical Report TUB-DAI 02/08-01, Technische Universität Berlin, DAI-Labor, Feb. 2008. http://www.dai-labor.de.

[24] D.-H. Shih, B. Lin, H.-S. Chiang, and M.-H. Shih. Security aspects of mobile phone virus: a critical survey. *Industrial Management & Data Systems*, 108:478–494, 2008.

[25] Symantec. Spyware.FlexiSpy, Mar. 2006.

[26] S. Töyssy and M. Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2):109–119, 2006.

[27] P. Wang, M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding the spreading patterns of mobile phone viruses. *Science*, pages 1167053+, April 2009.

[13] make it useless