A framework for on-device privilege escalation exploit execution on Android

Sebastian Höbarth
Upper Austria University of Applied Sciences
sebastian.hoebarth@fh-hagenberg.at

Rene Mayrhofer
Upper Austria University of Applied Sciences
rene.mayrhofer@fh-hagenberg.at

ABSTRACT

Exploits on mobile phones can be used for various reasons; a benign one may be to achieve system-level access on a device that was locked by the manufacturer or service provider (also known as 'jailbreaking' or 'rooting'), while potentially malicious reasons are manifold. Independently of the use case however, a specific exploit is not sufficient to achieve the desired access rights. Typically, exploits provide *temporary privilege escalation* immediately after their execution. To provide additional access to applications, *permanent privilege escalation* is required – in the benign case, including secure access control for the user to decide which (parts of) applications are granted elevated access. In this paper, we present a framework that can use arbitrary temporary exploits on Android devices to achieve permanent 'root' capabilities for select (parts of) applications.

1. INTRODUCTION

Current mobile phone platforms such as Android, iOS, or Maemo/MeeGo all build on native code for their respective kernels, libraries. User space utilities and services are therefore often susceptible to typical programming errors such as buffer overflows or missing input sanitization. These errors typically lead to application crashes or malfunctions, but can sometimes allow the caller of the corresponding service or function to cause unintended consequences.

One example for a common class of programming errors is a *buffer overflow* in native C code that allows user-provided (often binary) input to be copied into a some allocated buffer (typically an array of bytes or characters) without validating the sizes of input and buffer. If this buffer is allocated sufficiently close to some program memory that contains addresses to program code (such as function return addresses or library function mappings), than a carefully crafted input can overflow the boundaries of the buffer and overwrite these addresses with new program code contained in the user input. This new program code will then be executed within the context of the running service or function and, if this context is associated with higher access privileges than the user providing the input, it can be exploited to achieve *temporary privilege escalation*, forming a so-called *buffer overflow exploit* (cf. e.g. [3, 6]). Another common example is missing input sanitization, which allows to open, read, write, or execute files with higher privilege by exploiting a service or function that is supposed to be limited to a certain path or type of files but fails to verify this accordingly.

The range of exploits that have already been published for Android and other mobile phone platforms is manifold, and their specific attack vectors are significantly different. Therefore, it is difficult to unify the actual exploit codes into a common structure or a framework; it is often not even necessary, considering that exploit codes written in C are most typically very short (in the order of a few hundreds of lines of code). However, the steps that need to be executed after an initial temporary privilege escalation was achieved are often similar. To progress from temporary to *permanent privilege escalation*, the (mobile phone) system needs to be modified. This typically includes the installation of new binaries that allow controlled elevation of access rights.

A more typical approach for handling various exploits is therefore to collect them into a common framework for executing as many exploits as possible and, upon the first successful execution of any of these exploits, to install permanent means for system-level access within the temporary context returned by the exploit. Metasploit[1] is currently the most comprehensive framework for exploits and is widely used [7], but focuses on being executed on a standard desktop/laptop computer. At the time of this writing, there are initial ports of Metasploit to iPhone and Nokia N900 and experimental builds for Android, but all these require a mobile phone on which the user already has system-level access (i.e. 'root' on Android or 'jailbreak' on iPhone). In contrast to Metasploit, we focus on executing Android system exloits on the devices themselves, with the aim of achieving permanent privilege escalation on a single device (independently of the benign or malicious use case after reaching system-level access). With the exception of a few closed-source Android applications that bundle specific exploits with additional (but, between the applications, different) steps for advancing from temporary to permanent privilege escalation (most notably *Visionary+* for the HTC Desire HD and *z4root* for multiple Android devices), we are not aware of other publications on Android on-device system exploit execution frameworks. Other related work concerning Android exploits and exploit prevention is discussed within the respective sections.

[1] `http://www.metasploit.com`

IWSSI2011 12 June, 2011, San Francisco, US
.

In this paper, we first analyze the different layers of the Android security architecture (Section 2) and briefly describe examples for specific exploits (Section 3). Our main contribution is to present initial steps towards a framework for exploiting Android devices (Section 4). This framework is designed to be extensible in terms of various exploits that are tried one after another, but the subsequent steps only need to be implemented once for achieving permanent system-level access ('rooting' the Android device refers to the *root* user on standard Linux kernels, which is not restricted in any way when no mandatory access control methods are in use). Furthermore, our framework already includes some specific examples of using this system-level access to demonstrate the wide-ranging capabilities that can be achieved (Section 4.2).

2. ANDROID SECURITY ARCHITECTURE

The Android platform has been designed to allow the installation of potentially untrusted applications. This is different from the iPhone security model, where all applications need to be installed through the Apple store (unless the mobile phone is 'jailbroken' to allow the installation from different sources) and are (supposedly) verified concerning their internal behavior prior to their publication. Therefore, the Android platform implements security mechanisms on different layers: application sandboxing as a high-level concept makes use of filesystem access control as enforced by the Linux kernel and permissions granted upon installation time to – selectively – pass the boundaries of these sandboxes. Applications are also cryptographically signed, but these signatures only provide some level of auditing and no security-relevant validation procedures before publication. In the following, we will describe these security layers and their potential shortcomings in more detail.

Android is based on a standard Linux kernel with minor modifications (e.g. an additional shared memory implementation and automatic device suspend handling) and thus inherits the Discretionary Access Control (DAC) on the filesystem level, which is based around user IDs (*uid*) and group IDs (*gid*). On top of the Linux kernel, Android uses custom user space libraries and services different from standard GNU/Linux user space tools. Applications can be developed in Java and compiled to the custom Dalvik byte code to be executed by a Just-in-Time (JIT) compiler and virtual machine on the device or using C/C++ code that is called from Java using the Java Native Interface (JNI).

2.1 Application Sandboxing

Applications installed on an Android system are confined to *sandboxes* that are defined by a unique uid (and matching gid) and which are created dynamically during the installation using the Android application manager. User and group names are equal and start with the '`app_`' prefix followed by an automatically incremented counter. The result is that all applications are installed using separate user and group identities and – using filesystem access control – are fully separated from each other and only have limited (typically read-only) access to system files and services. Calling functions or services outside this sandbox requires the use of specific APIs which are restricted by permissions granted at installation time (cf. Section 2.3).

The sandbox restrictions are enforced by the kernel (and respective user space services/daemons) and therefore apply to all applications, including native code called directly via JNI or indirectly using the `exec` system call. If applications intend to share data, i.e. to cross the boundaries of two or more sandboxes on the filesystem level, the same shared uid needs to be added to the application manifest, which is only possible when all applications are signed with the same private key (cf. Section 2.4). The uids and permissions according to the package manifests of all applications installed on a device are stored in `/data/system/packages.xml` and are read-only accessible to all applications.

Recently, it has been shown that the Android sandboxing concept has a fundamental flaw that allows transient privilege escalation based on calling services offered by other application which were granted more permissions than the calling application [2].

2.2 Filesystem Access Control

The filesystem DAC on Android uses the traditional Unix permissions. When an application stores data, file permissions are by default set to '`rw-rw---`' (0660 in octal notation). That is, applications installed with a different uid and gid can neither read, write, nor execute the files. Files can be made publicly accessible (i.e. by different uids and gids) using the `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` flags when creating them using the Android Java APIs or with the `chmod` system call in native C/C++ code. As defined in the standard Unix access control model, this is at the discretion of the application respectively uid creating the file (more specifically, the uid having write access to the directory in which the file is created).

By default, the application data home directory is located at `/data/data/<package name>/` and can contain the following directory structure:

databases is the default location for sqlite databases

libs contains all native libraries of the application, copied during the installation process

files is the default directory for all files created by the application itself at run time

shared_prefs contains the XML based shared preferences for the application

Pre-installed system applications with higher privileges such as `com.android.phone` typically also use the same directory hierarchy. However, these default paths may not be used on all devices; for example, Samsung devices use `/dbdata/databases/<package name>/` for pre-installed applications. It is therefore necessary for any exploit framework to be flexible in terms of filesystem layout and be aware of different default locations for different devices.

During system bootup, parts of the filesystem are mounted with different options. The directory `/data` is mounted by default with the 'rw,nosuid,nodev,relatime' options, which means that files with the so-called 'setuid' bit are not honored. They are executed only with the permissions of the calling uid instead of being granted the permissions of the owner of the respective file (which will often be root for files with 'setuid' set). By remounting this partition with

```
1 mount -o remount,suid,rw /data /data
```

the 'setuid' bit will be honored. That is, executing the respective file will grant the *effective* uid of the file owner

and all associated permissions. The directory `/system` is mounted by default with the 'ro,relatime' options, which adds the constraint of making the whole partition read-only (note that `/system` is not mounted with the 'nosuid' option, because the Android system also uses 'setuid' binaries under this location). In order to add or change files under `/system`, we can remount the partition with

```
mount -o remount,rw /system /system
```

for full filesystem access.

On some devices made by HTC (e.g. the HTC Desire or HTC Desire HD), the `/system` partition is additionally protected with a 'NAND lock' (when the '@secuflag' security flag set to 'S-On'), which is implemented by the baseband processor that manages the GSM radio and moderates access to the NAND flash. The boot loader and radio ROM together define the state of the NAND lock and, after the Linux kernel has been loaded and executed, some NAND partitions are therefore no longer writable to the Linux kernel when the flag is set to 'S-On'[2]. Most notably, all changes to the `/system` partition are prevented from being written to the underlying flash memory (and result in different system call error codes) even when the partition has been remounted with read-write access from the kernel point of view. That is, permanent changes are prohibited while temporary runtime privilege escalation spanning the boundaries of sandboxes is still possible. To work around this restriction, either the NAND lock needs to be removed (also referred to as 'S-Off') by changing the boot loader and/or radio ROM[3], or the modifications need to be done in recovery mode[4], which is not confined to NAND lock by the boot loader.

2.3 Permissions

In order to leave the sandbox in terms of filesystem access or calling functions or services outside the own (group of) application(s), additional capabilities are needed. The requested permissions are displayed at application installation time and the user needs to accept all permissions for the application to be installed. Unfortunately, it is not yet possible to grant only some permissions and reject others, which would require the Android platform to check and potentially query the user for permissions at run time instead of only at install time.

For access to the restricted APIs, applications must include permissions such as

```
<uses-permission android:name="android.
    permission.INTERNET"/>
```

in the project `AndroidManifest.xml` file. If an applications tries to access some feature without declaring the associated permission, the attempt will fail and a `SecurityException` will be thrown.

In some cases, there are possible workarounds for this permission system. For example, it is possible to send and receive data from the Internet without the `INTERNET` permission by using *Intents* to start the browser in a security context that supports the `INTERNET` permission:

```
startActivity(new Intent(Intent.ACTION_VIEW,Uri
    .parse("http://www.google.com/upload?imei="
    +imei)));
```

To receive the queried data within the custom application, it simply registers an *Activity* in the `AndroidManifest` with appropriate *Intent-Filters*:

```
<activity android:name=".ReceiveActivity">
<intent-filter>
<action android:name="android.intent.action.
    VIEW"/>
<category android:name="android.intent.category
    .DEFAULT"/>
<category android:name="android.intent.category
    .BROWSABLE"/>
<data android:scheme="response" android:host="
    download"/>
</intent-filter>
</activity>
```

With this configuration, the Android platform will automatically start the activity 'ReceiveActivity' upon receiving data from the Internet (which may or may not be visible to the user depending on the speed of the device in terms of starting and closing intents). Then, data can simply be queried with:

```
getIntent().toURI();
```

which contains the full data object. Note that transient privilege escalation exploits like this one are currently a fundamental problem of the Android permissions architecture [2] and will therefore likely need to be fixed in a major platform revision (as opposed to minor bug fixes distributed within over-the-air upgrades). Exploiting these issues is orthogonal to our approach, because it allows to indirectly elevate permissions without exploiting other programming errors. In our framework, after successfully executing any temporary privilege escalation exploit, an application will receive *complete* system-level access and therefore all capabilities without explicitly declaring them in its manifest.

2.4 Application Signing

Each application has to be signed by a private key with an associated certificate, as unsigned applications cannot be installed on the device. However, this certificate may be self-signed or signed by any certificate authority (CA). The only security benefit of signing is therefore – with current Android security policies – auditing and non-repudiability: assigning applications to their developers. Within the application signing approach, it would be possible for future versions of Android (or specific vendor variants) to restrict installation to applications signed by specific CAs (similar to the Apple iPhone/iPad security model). As mentioned above, only applications signed by the same private key may request the same uid upon installation, which is not relevant for our exploit framework.

2.5 Exploit Prevention

Current desktop/laptop and server operating systems implement multiple techniques towards preventing (or at least significantly complicating) the exploitation of some classes of programming errors. Most of these security measures focus on buffer overflows and return code or library function

[2]See e.g. `http://tjworld.net/wiki/Android/HTC/Vision/BootProcess` for more details (last retrieved 2011-03-23).

[3]The AlphaRev (`http://alpharev.nl/` and unrevoked (`http://unrevoked.com/`) tools use this approach at the time of this writing (last retrieved 2011-03-23).

[4]The 'recovery' image is a separate combination of kernel and initramfs image that can be called by the boot loader when pressing the respective key combinations during device bootup.

manipulation in native code and include, among others, so-called 'canaries' (verification values before and after return code buffers) in user space libraries, Address Space Layout Randomization (ASLR) techniques to impede guessing correct function memory addresses, and making executable and writable areas in system memory mutually exclusive (i.e. preventing memory regions that can be executed from being modified, and vice versa).

Unfortunately, Android does not currently implement any of these security measures, partially because most code is protected by the Java virtual machine (Dalvik) and (presumably) for performance reasons. Recent work shows that ASLR would be possible in Android with minimal performance impact and reasonable (but not excellent) entropy [1]. We are not aware of work towards porting other measures for preventing additional classes of exploits to Android (e.g. the PaX kernel patches).

3. EXAMPLE EXPLOITS

For reaching the aim of (temporary and subsequently permanent) privilege escalation, an applications needs to break the boundaries of its sandbox, ideally to change its security context to that of the *root* user. As with other Linux distributions that do not implement additional Mandatory Access Control (MAC) methods such as SELinux [5] (applying SELinux to Android has already been proposed, but not yet implemented in the standard Android tree [9]), Smack [8] (which will be used as part of the security infrastructure of the upcoming MeeGo version 1.2), TOMOYO [4], or AppArmor, the root user is completely unrestricted and has full system-level access. Within the normal Android security architecture, there is no possibility for an application to achieve root access permissions. The consequence is that only by exploiting programming errors in system code that is executed with a root security context, applications can achieve this privilege escalation.

In this section, we briefly describe four example exploits that provide temporary escalation to root privilege to the executing application. All presented exploits in this chapter are created with the Android Native Development Kit, and their applicability depends on the Android system version (known exploits tend to get fixed in updated version). However, because not all manufacturers provide the latest firmware updates for all their devices, some of these (and other) exploits remain open for significant periods. For some exploits, it is necessary that the Android Debugging Bridge (ADB) is activated on the device.

3.1 Missing input sanitization

One of the oldest published Android exploits is a classical case of a service not correctly sanitizing (i.e. validating) input it receives from potentially untrusted sources. Up to Android version 1.6, the user-space device management daemon `udev`, which handles adding or removing device nodes and firmware loading, fails to verify which process requests to install new firmware. The exploit itself is triggered in three steps: First, three files required for the fake firmware installation are created in the home directory of the application or as shell user in the `/data/local` directory:

loading is an empty file that will be used by firmware installation tools for reporting status updates.

data is a symlink to `/proc/sys/kernel/hotplug`.

hotplug contains the full path to the file that should be executed by the `init` process. In our case, the hotplug file would contain the path to the exploit framework binary (cf. Section 4), which will subsequently be executed in the root security context.

Second, a netlink datagram socket connection is opened (from any user space application) to the `init` process (which, by convention on Linux systems, uses the initial process id, i.e. pid=1). After this connection has been established, the application sends the respective keywords to add a firmware to the system along with the path to the files created in the first step. This causes the process to start adding a new firmware without validating the caller's permissions and copies the content of the `hotplug` file into the `data` file, changing the system hotplug binary to point to our exploit framework binary.

Third, we simply trigger a hotplug event, e.g. by turning WiFi on or off, with the effect of invoking `/proc/sys/kernel/hotplug` – and therefore our own binary – with system (root) privileges.

3.2 Overflowing limit of available processes

The goal of this exploit is to overflow the supported number of processes (defined by `RLIMIT_NPROC`) created by the same uid (2000, the `SHELL` user). If this process limit has been exceeded, no new processes will be created by the Linux kernel for this uid and subsequently the debugging daemon `/sbin/adbd` normally started in the context of the `SHELL` user will be killed by its respective parent. The `adbd` process is marked for autostart and will therefore be restarted by the system. In the default case, the `adbd` process is started with root privileges and then changes its uid to 2000 (dropping privileges). However, in an exploited system, this is no longer possible because the maximum number of processes for this uid has already been reached; the uid change fails and `adbd`, failing to properly deal with this error, remains running with uid 0. Connecting to the debugging shell offered by `adbd` will therefore automatically grant root privileges. According to our tests, this exploit works until the Android version 2.2.

3.3 Remapping shared memory

This exploit tries to change the global system settings that effect the system shell. An *ashmem* ('Android shared memory') area is owned by the `init` process and holds references to the shared memory areas and as well as system attributes. The virtual file `/proc/self/maps` lists all these memory maps of the current process and is parsed by the exploit to locate the `/dev/ashmem/system_properties` area in the shared memory, which it subsequently tries to remap using the POSIX `mprotect` function. Until Android version 2.2, the ashmem implementation fails to prevent remapping shared memory regions by unprivileged processes. Thus, after remapping the shared memory region, the exploit only needs to locate the `ro.secure` attribute and set to 0. Finally, the `adbd` process is restarted so that the settings are applied and all subsequently opened debugging shells are started with root privileges.

3.4 Restricting access to ashmem

The goal of this exploit is to restrict the *ashmem* shared memory where system properties are stored. As mentioned

in the previous exploit, the `adbd` process relies on the ability to read the `ro.secure` property to determine whether to change its uid or to retain root privileges. If `adbd` can not read the properties from shared memory because the process can not map the ashmem page, then it will – erroneously – not drop its privileges under the assumption that `ro.secure` is 0. The exploit reads the environment variable `ANDROID_PROPERTY_WORKSPACE` which contains the size of the property area and maps the memory again. Then, the ashmem protection mask will be set to 0 so that no other process can read the properties, including `adbd` and therefore triggering the erroneous privilege escalation. The side effect of this exploit is that other processes will also be excluded from reading this memory area, affecting the whole system until the next reboot. This exploit has been verified to work until Android version 2.1.

4. FRAMEWORK

After achieving (temporary) system privileges and therefore working around the current Android security measures, applications are no longer restricted in any way (in the absence of a kernel MAC implementation or kernel-level virtualization). In many cases, Android devices are being deliberately 'rooted' (i.e., applications being installed with root privileges) to get access to system files (such as those under the `/proc` and `/sys` virtual filesystems), e.g. to use these privileges to optimize the energy consumption of the device or to overclock the processor. These special applications must always be able to access the respective files and services and it would not seem acceptable if every application had to exploit the system for every system-level access. Furthermore, some exploits may cause side effects until the system is fully rebooted (such as broken DNS resolution). For all these cases, the device should be permanently rooted, i.e. the temporary privilege escalation should be transformed to a permanent privilege escalation, which is one of the main aims of our framework and described in more detail below.

Our exploit framework (available at `http://openuat.org/android-exploit-framework`) can be applied independently of any specific exploits, allowing experimentation with different exploits before using it to achieve permanent privilege escalation. As described in section 2.2, there are also devices where permanent rooting is (currently) not trivial, since the file system is especially protected by kernel modifications to prevent changes to the `system` flash area during run-time. For these situations, the framework also includes some additional functions explained in section 4.2.

4.1 Permanent root privileges

Achieving permanent privilege escalation requires working around the Android file system access control (cf. section 2.2) — the temporary privilege escalation already disabled the sandboxing (cf. section 2.1) and permissions (cf. section 2.3) security measures. To achieve this goal, there are several possibilities which are independent of the exploit used to gain temporary system-level access. The straightforward approach is to rely on basic UNIX file system permissions and install a new binary with 'setuid' permissions and owned by the root user (e.g. a shell that allows executing arbitrary other commands or the 'SuperUser' Android application with an adapted 'su' binary installed as `/system/bin/su` that asks the user for confirmation when another application tries to use it to gain root permissions).

On filesystems where the `/system` filesystem is NAND locked and it can therefore not be modified permanently while the respective kernel is running, the system would have to either: a) modify the `recovery` flash area to embed a new startup script (described in more details below), trigger a reboot into recovery mode where the device boot loader executes this `recovery` instead of the normal `boot` kernel and initramfs image, and exploit the fact that the recovery system is booted without NAND lock and that the new startup script can therefore modify the `/system` filesystem permanently; or b) modify the `boot` flash area to e.g. set the `ro.secure` system property to 0 (in the same way as modifying `recovery`) and trigger a normal device reboot so that the modified initramfs image is loaded by the kernel and, during each subsequent device boot, applies the changed settings. An automated process that uses either of these modes of operation has not yet implemented in our framework but is subject to future work.

Currently, our framework – implemented as a small binary in native code – executes the following steps when called with temporary root permissions:

1. It reads the currently mounted file systems from `/proc/mounts` and remounts the `/system` filesystem with read-write permissions (using direct system calls instead of the standard user-space tools). If this remounting fails, the framework aborts, as the subsequent steps can not be performed due to `/system` being NAND locked to the kernel. However, the framework can alternatively change the access permissions for system databases (cf. section 4.2), so that at least these files can be read by arbitrary applications. The disadvantage is that, without additional 'setuid' binaries with an user interface, this access can not be made visible to the user and is therefore only relevant to the more malicious use cases.

2. The framework itself is copied into the `/system/bin` folder. Should this attempt fail (another potential symptom of the partition being NAND locked), the alternative action is to set the 'setuid' bit on the built-in shell (`/system/bin/sh`), which grants root permissions to all applications calling the shell until the next system reboot. This secondary approach is possible because some implementations of the NAND lock protection seem to be faulty: Although creating and/or writing to files or directories under `/system` is prevented, modifying file meta information (such as the 'setuid' bit) returns an error on the system call but will still be performed in the YAFFS2 filesystem memory structures and therefore persist as long as the kernel is running and the filesystem is not being remounted.

3. If the copy process of the framework has been successful, the framework 'setuid' bit is also set on the framework binary, which leads to the fact that the root privileges are now permanently available.

For NAND locked devices, the following steps would be necessary to embed the modifications into the `recovery` or `boot` flash areas before triggering a reboot into the respective (recovery or normal bootup) mode, where the above changes can be executed without the NAND lock preventing the remount or copy system calls:

1. It has to be discovered which block device file refers to the `recovery` or `boot` partition of the device. This can

easily be done by parsing `/proc/mtd`. For example, on a (NAND locked) HTC Desire the default recovery and boot devices are `mtd1` and `mtd2`, respectively.

2. The contents of the respective file under `/dev/mtd/` are copied to the micro SD card and the embedded kernel and initramfs image are extracted[5].

3. The extracted initramfs image is a gzipped cpio archive that the kernel unpacks during bootup to form the root filesystem residing in a RAM disk and therefore making all changes within the root filesystem temporary until the next reboot. Within the main boot script `init.rc`, the global system attribute `ro.secure` can be changed from 1 to 0 (cf. section 3.3).

4. In the last step, all steps from 1–3 are performed backwards, so that the manipulated boot image replaces the original one.

Note that these steps are often followed when custom ROM images are created for specific devices, e.g. to add additional features or the SuperUser application to the main system image. However, they are typically done on a laptop/desktop to the effect of creating a new image for the `recovery`, `boot`, and/or `system` flash areas. These images are then bundled in `update.zip` files that can be executed by the standard device boot loader to update the respective flash areas. In contrast, our framework will be extended to perform these modifications on the device itself without resorting to additional host systems. To the best of our knowledge, this has not been presented publicly before.

4.2 Additional features

As mentioned in the previous section, it is not always possible to permanently achieve system-level permissions. For these cases, the framework supports making changes to the `/data` (instead of the `/system`) filesystem, which remain permanent even on NAND locked devices. Specifically, database files of various applications will be made available to all processes, specifically:

accounts.db contains account information used by various applications.

EmailProvider.db contains all relevant data about the used mail account and the mails itself.

settings.db contains specific system settings such as screen timeout.

gmail.db contains the standard Google Mail account that is used on the system.

The second additional feature of the current framework version is a keylogger, which records all registered hardware button events into a custom log file.

5. DISCUSSION AND FUTURE OUTLOOK

In this paper, we have reviewed the Android security architecture and its specific security measures with regards to achieving system-level permissions using privilege escalation techniques during run-time. Four specific exploits that were previously published by other authors have been implemented within a common structure to temporarily escalate the privileges of the calling process to *root* level. Our framework then transforms this temporary to a permanent privilege escalation while keeping the required modifications to the Android system to a minimum (only installing additional binaries with their 'setuid' bit set) and being easily reversible. This framework can be used with arbitrary current and future exploits and is applicable to all current Android devices. Some current devices feature a so-called 'NAND lock' to further protect their main filesystem, which requires additional steps and at least one device reboot to work around. We have described a potentially automated execution of these steps on all currently available devices, although their implementation is subject to future work. Furthermore, we intend to include additional helper functions and binaries for specific tasks such as parsing and modifying system database, password stores, or inspecting and modifying network traffic in future versions of our framework.

With the release of our framework, we hope to demonstrate that the Android security architecture is not – at the time of this writing – sufficient for preventing malicious applications from gaining full system-level access. Possibilities for improving the security of Android devices are manifold; the most pressing improvements would be the inclusion of ASLR (address space layout randomization) and NX (no-execute) patches in the Android kernel, more fine-grained application capabilities (e.g. to distinguish which network protocols and resources/URLs applications may use instead of giving them full network access), and some form of MAC (mandatory access control) to more thoroughly restrict application permissions on the kernel level and therefore to harden the application sandboxes that currently rely only on filesystem DAC (discretionary access control).

[5]The layout of these images is specified in `http://android.git.kernel.org/?p=platform/system/core.git;a=blob;f=mkbootimg/bootimg.h`.

6. REFERENCES

[1] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev. Address space randomization for mobile devices. In *Proc. WiSec 2011*. ACM Press, 2011.

[2] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proc. ISC 2010*. Springer-Verlag, 2010.

[3] J. C. Foster, V. Osipov, N. Bhalla, and N. Heinen. *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Syngress Publishing, 2005.

[4] T. Harada, T. Horie, and K. Tanaka. Towards a manageable Linux security. In *Proc. Linux Conference*, 2005.

[5] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX '01*, 2001.

[6] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(3):20–27, July-August 2004.

[7] E. Ramirez-Silva and M. Dacier. Empirical study of the impact of metasploit-related attacks in 4 years of attack traces. In *Proc. ASIAN'07*. Springer-Verlag, 2007.

[8] C. Schaufler. Smack in embedded computing. In *Proc. Ottawa Linux Symposium*, 2008.

[9] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy*, 8:36–44, 2010.