Master System and Network Engineering

University of Amsterdam

Research Project 2

Android patching

From a Mobile Device Management perspective

Author: Cedric Van Bockhaven
cedric.vanbockhaven@os3.nl

Supervisor: Jochem van Kerkwijk
JvanKerkwijk@deloitte.nl

June, 2014

Abstract

Android is currently the most popular smartphone OS in the world. Many different devices with outdated Android versions and kernels pose a risk as they become a potential target for attackers, making enterprises reluctant to allow employees to bring their Android devices to the workplace.

Fixing vulnerabilities by patching the kernel and Android runtime in-memory allows leaving the underlying system mostly untouched while providing protection against emerging threats. The proposed techniques allow to shift the responsibility of bringing out patches from the mobile device vendor to the MDM solution.

Acknowledgements

I would like to thank the people at Deloitte Risk Services, where I conducted my research, for their insights and support: my supervisor Jochem van Kerkwijk, Werner Alsemgeest, Joost Kremers, and Thomas Bosboom. They provided a nice work environment and Android devices to test my research on.

Contents

1 Introduction

Enterprises allow employees to bring their own device to the workplace under a Bring Your Own Device (BYOD) policy. These devices are usually smartphones and are hooked onto corporate data or corporate infrastructure. More and more Mobile Device Management (MDM) solutions emerge that can monitor and control these devices.

Of all smartphone owners in the world, about 80% has an Android device [1]. In stark contrast to this number stands the Android market share for enterprise environments where people are allowed to bring their own device, which is only 26% [2].

The reason for this significant difference can be found in a broad panoply of different Android devices and versions. These systems often contain outdated (vulnerable) Android versions, and do not always receive updates, which make them an additional security and data leak risk for the company.

Google's Android relies on the vendors of the devices to push updates. In many cases, vendors do not bring out updates, do not offer over-the-air updates, or drop support for the device. It is then up to the user to manually conduct a firmware update, which is not always in reach for non-technical people.

In June 2014, Gartner published a comparative report, dubbed the "Magic Quadrant for Enterprise Mobility Management Suites" in which MDM systems were evaluated according to the current needs of the business world [3]. The report also stated the importance and ever growing need for Android support and data loss prevention (DLP) in the MDM world.

This research investigates if it is feasible to patch the Android operating system through the MDM solution when new security vulnerabilities are discovered. There is no longer any need to rely upon the vendor to push updates as this responsibility is shifted to the MDM.

2 Problem statement

One of the reasons why Android has become such a success and has a world wide adoption, is that other companies, besides Google, can also benefit from the same technology as the operating system has been open-sourced. Android has become the operating system of choice for many smartphone manufacturers. Adapted versions of the Android operating system are being made by these vendors to support different device models, and usually include a modified kernel and extra drivers to support the hardware. Whenever an Android device is released onto the market, it ships with a certain version of the Android OS.

Even though the Android OS is maintained by Google, the responsibility of bringing updates to the end-user devices is in the hands of the mobile device vendors. Google does not push updates to devices of other mobile device vendors. However, not many other device vendors actually bring out updates, or offer over-the-air updates. All vendors have a limited support window in which they bring out updates. This, combined with little update initiative from the end users, leads to a slow adoption of newer Android versions and a spectrum of different versions and builds that can be found in the wild, as illustrated by Figure 1. Android 2.3 Gingerbread, which was released in December 2010, still accounts for 20% of the Android version distribution anno 2014.

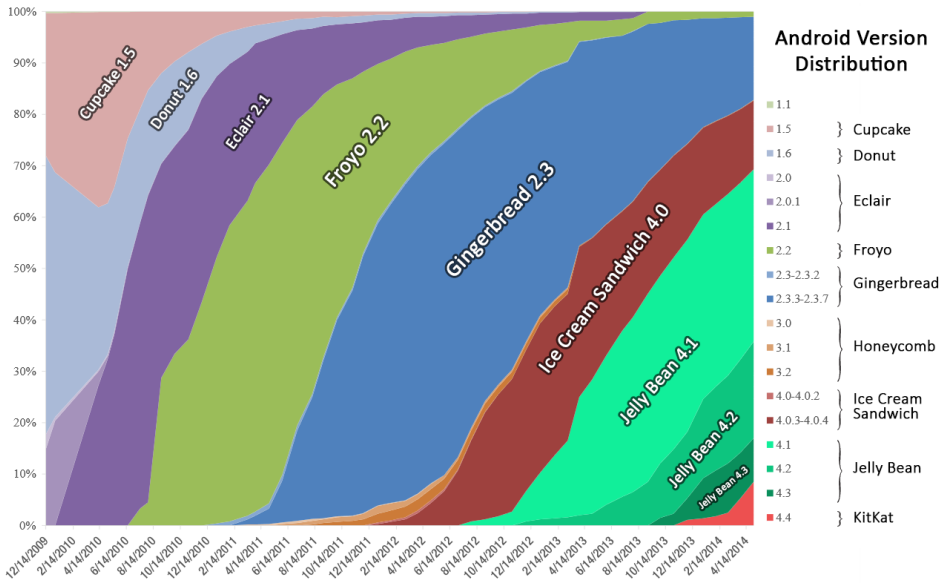

Figure 1: Android version distribution
Source: Adopted version of Fjkumar's original, CC BY-SA 3.0

The story is different for iOS, where the maker of the OS and the firmware is also the vendor of the device. The amount of different devices that have to be supported is significantly lower. Updates can be easily pushed over-the-air in this case.

All these different Android devices, firmware builds and kernels are consequently prone to security vulnerabilities when they become outdated. They may then become the target of malevolent parties that may exploit these vulnerabilities (either as part of a targeted or non-targeted attack). Risks include the theft of company secrets, unprivileged access to company equipment, and sensitive data exposure, which may all lead to financial losses.

In an enterprise environment, this may explain why employers are reluctant to adopt Android devices in their BYOD policies. It is hard to support all the different devices and many could introduce security risks for the company, such as data leaks. There is a significant difference in the global Android market share (80%) and in enterprises (26%) [1, 2].

Moreover, Android devices with vulnerabilities can often be rooted to obtain unrestricted access to the lowest device levels. This makes the devices unpredictable as it is no longer possible to warrant the integrity of the device. Using privileged root access, security measures can be easily disabled or circumvented. The integrity and confidentiality of the MDM and company data can no longer be warranted from that moment on.

All of the above demonstrates the need for an out-of-band update mechanism that no longer relies on the vendor. The MDM solution could provide security updates for the device in order to maintain a protected environment. An Android agent application that is installed on the user's device could be created that performs patches, regardless of the underlying hardware and OS version.

2.1 Research question

Given the arguments in the previous section, an out-of-band update mechanism for mobile devices is desired. The main research question is therefore:

- In an environment where outdated devices have to be considered, such as workplaces with a BYOD policy; is it possible to patch security vulnerabilities in Android devices through the MDM?

Subquestions that arise during the conduction of the research are:

- How can patches be created (and applied) for the different system architectures in an easy fashion?

3 Related research

The most relevant research on patching Android devices was conducted by Mulliner et al. [4] in 2013. Their solution, PatchDroid, describes in-memory patching of both native and managed code. However, kernel-level vulnerabilities were not considered in this paper.

In 2010, Enck et al. [5] published a paper on TaintDroid, an information-flow tracking system for Android by using variable-level tracking within the Dalvik VM. This research was followed up by Sarwar et al. [6], who prove that the effectiveness of TaintDroid is not airtight. A similar system was developed by Egele et al. [7] for iOS, called PiOS.

In March 2014, Ho et al. [8] presented the Practial Root Exploit Containment (PREC) framework. PREC can dynamically identify system calls from high-risk components such as third-party native libraries, and execute those system calls within isolated threads. Hence, PREC can detect and stop root exploits.

In May 2014, Joost Kremers [2] completed his master's thesis on Mobile Device Management with relation to the Android OS. He provides a framework that allows to perform an evaluation of the implementation of the MDM system. This framework is based on Keunwoo Rhee's research and is therefore dubbed the "Extension of Rhee's framework" (ERF) [9, 10].

4 Primer

In the current section, the architecture of the Android OS will be discussed as an introduction to patching the components that the OS consists of.

4.1 Android architecture

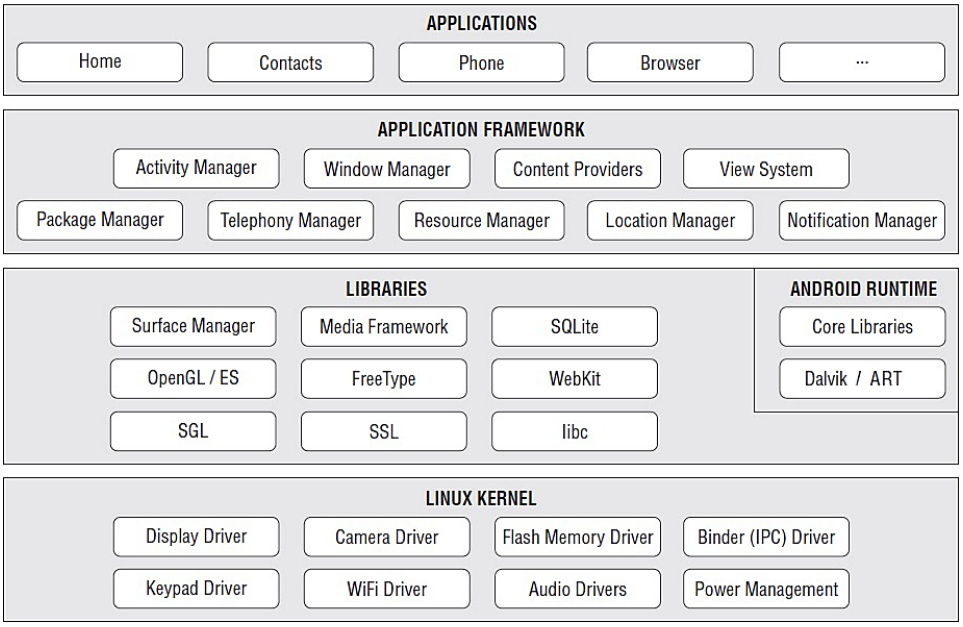

Figure 2: Android architecture
Source: Selva Kumar, http://goo.gl/k44A5j

In the highest layer of the Android architecture, everyday Android applications can be found such as the ones that can be obtained from the Google Play store. Applications then hook into application framework functions. These API functions can for instance be used to obtain the current Wi-Fi status or send text messages.

The applications and application framework are programmed in Java and subsequently translated into an intermediary bytecode at compile-time that can then be interpreted by the runtime.

At the moment, the Dalvik VM is the default runtime, which is soon to be replaced by the Android Runtime (ART). ART mainly features ahead-of-time compilation, but is still in development and only available in Android 4.4 KitKat.

4.1.1 Dalvik runtime

Dalvik is a process virtual machine in which Android applications run. The Dalvik VM runs executables in DEX (Dalvik Executable) format. The DEX format is designed for systems that are constrained in terms of memory and processor speed. Dalvik was originally authored by Bornstein [11] and is open-source.

Android applications are mostly written in Java. The bytecode that is generated by the Java compiler has to be converted to Dalvik bytecode. The Dalvik bytecode is stored in DEX and/or ODEX (Optimized DEX) format. Since Android 2.2, Dalvik has a just-in-time (JIT) compiler [12].

Zygote is the Dalvik VM master process. Zygote is responsible for starting and managing application processes. It preloads the shared libraries and forks off the application processes.

The Dalvik VM does not sandbox the applications: any application can run native code by embedding native libraries into the APK. The individual capabilities/permissions of the application are enforced by the Linux kernel [13].

4.1.2 ART runtime

In Android 4.4, a new experimental runtime was introduced that will eventually replace Dalvik: the Android RunTime (ART).

ART features ahead-of-time (AOT) compilation. When an application is installed on a device running ART, it is compiled on the system itself. It generates native code from the Dalvik bytecode (DEX).

This way, ART can profit of instructions that are specific for the CPU, and ultimately gain performance wins. Disadvantages may include the need for more storage space and a longer installation time.

4.1.3 Android kernel

The Android kernel is largely based on the Linux kernel. It has architectural changes that are implemented by Google outside the typical Linux kernel development cycle, such as the inclusion of components like Binder, ashmem, pmem, logger, and wakelocks [14].

Operating systems provide different levels of access to resources, which are also referred to as *privilege rings*. The privileges are generally hardware-enforced by the CPU at hardware or microcode level. Rings are arranged in a hierarchy from most privileged (ring 0) to least privileged. On Android (and Linux) systems, ring 0 refers to the kernel, whereas ring 3 refers to user space applications. In between, device drivers can be found that may have more privileges than user space applications, but are more restricted than the kernel. [15]

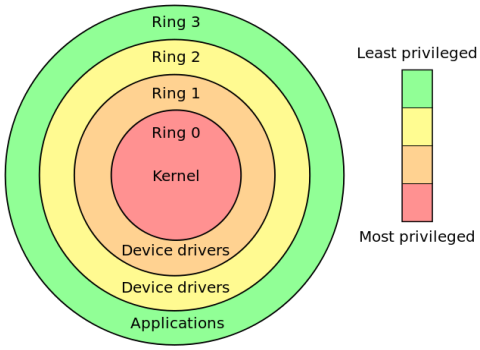

Figure 3: Privilege rings
Source: Hertzsprung, CC BY-SA 3.0

5 Hooking

Android device owners are generally not given root access to the operating system by the vendor. However, root access can be obtained by exploiting security flaws in Android, which is used frequently by the open-source community to enhance the capabilities of their devices, but also by malicious parties to install malware [16]. In certain cases, vendors allow rooting of their devices and offer the methods to do so [17].

Framework and kernel vulnerabilities may exist for Android devices, and methods will be described to patch vulnerabilities for both. This is usually done by hooking the original functions and replacing them with a patched version.

5.1 Runtime hooking

Framework vulnerabilities can usually be patched by hooking into the vulnerable functions through the runtime. This is why the terms "runtime hooking" and "framework hooking" will be used interchangeably. The two earlier described runtimes, Dalvik and ART, are the only ones in use on Android at this moment. Hooking techniques for both will be discussed in this section.

5.1.1 Dalvik hooking

Solutions are already available that hook into the Dalvik VM, open-source as well as closed-source. Since Dalvik hooking has been researched in the past, only an overview of the available solutions is given:

PatchDroid: DDI & ADBI

The Dalvik Dynamic Instrumentation (DDI) toolkit, which implements the concepts found in the PatchDroid paper [4], can hook dynamically into the Dalvik VM and patch framework functions.

The Android Dynamic Binary Instrumentation (ADBI) toolkit is based on library injection and hooking function entry points (in-line hooking). It heavily relies on `ptrace`.

PatchDroid consists of a daemon, `patchd`, which is launched at system startup. It monitors the system for new processes in order to apply any necessary patches prior to process execution. Syscalls as well as framework functions can be hooked using the PatchDroid toolkits. Both are open-source.

Xposed framework

The Xposed framework [18] modifies the `/system/bin/app_process` executable on Android to load a JAR file on application startup. The classes in that JAR file will be loaded into every application process, allowing to extend and override methods from the base framework. Xposed is open-source.

Cydia Substrate

Cydia Substrate, developed by *saurik*, allows modifying and hooking framework functions by preloading itself into all of the spawned Android application processes, much like the Xposed framework does. Substrate is closed-source.

As a proof of concept for the PatchDroid paper [4], Rekey is an Android application that was developed to fix the Master Key vulnerability. The Master Key vulnerability exploited a bug in the `ZipFile` class which could be used to bypass signature verification of APK files, eventually leading to root access [19].

5.1.2 ART hooking

ART is a new runtime for which no hooking system currently exists. ART is experimental and currently completely undocumented.

On devices with ART support, the `dex2oat` application compiles the DEX code found in APKs to native code. After enabling ART, a file `boot.oat` containing the framework classes, will be compiled upon reboot. Just like the Dalvik VM preloads the framework jars, this file is preloaded in all Android applications under ART.

Hooking into ART could involve patching the framework code in-memory, or preloading another library before `boot.oat` that hooks into the framework functions. Binary patching the `boot.oat` file, or recompiling with changed framework classes is a more permanent solution.

In the compilation command we can find all framework libraries as they would be found in the `BOOTCLASSPATH` of the Dalvik VM:

```
/system/bin/dex2oat
--image=/data/dalvik-cache/system@framework@boot.art
--runtime-arg -Xms64m --runtime-arg -Xmx64m
--dex-file=/system/framework/core-libart.jar
--dex-file=/system/framework/conscrypt.jar
--dex-file=/system/framework/okhttp.jar
--dex-file=/system/framework/core-junit.jar
--dex-file=/system/framework/bouncycastle.jar
--dex-file=/system/framework/ext.jar
--dex-file=/system/framework/framework.jar
--dex-file=/system/framework/framework2.jar
--dex-file=/system/framework/telephony-common.jar
--dex-file=/system/framework/voip-common.jar
--dex-file=/system/framework/mms-common.jar
--dex-file=/system/framework/android.policy.jar
--dex-file=/system/framework/services.jar
--dex-file=/system/framework/apache-xml.jar
--dex-file=/system/framework/webviewchromium.jar
--oat-file=/data/dalvik-cache/system@framework@boot.oat
--runtimearg -implicit-checks:none --instruction-set=arm
--instruction-set-features=default --base=0x70000000
--image-classes-zip=/system/framework/framework.jar
```

Efforts are being made by the Xposed developer *rovox89* [20] to make the Xposed framework compatible with ART. Since ART is a work in progress, and only a preview is available to developers, Xposed will be updated once there is a stable codebase.

5.2 Kernel hooking

Occasionally, security vulnerabilities are found in the Linux kernel. Since the Android kernel is largely based off the Linux kernel, the same vulnerabilities are also introduced on Android. These weaknesses can be abused to gain illicit privileged access to the system, such as root and even ring 0 access. Among the exploitable bugs that can be observed, overflow vulnerabilities, race conditions, dereference vulnerabilities, and buggy drivers can be found. Popular solutions that provide live patching of kernel vulnerabilities often rely on the use of a Linux kernel module (LKM), as explained in the following sections.

The following solutions could be used to patch kernel vulnerabilities:

- Binary in-memory kernel vulnerability patches such as with Ksplice using an LKM (section 5.2.1).
- Hooking addresses of variables and functions that reside in kernel space such as with Kprobes using an LKM (section 5.2.2).
- An approach will also be introduced to dynamically patch functions in-memory by exploiting the same vulnerabilities as the ones that have to be patched. This approach, dubbed *expatting*, aims to provide device-independent patching without the need for an LKM (section 5.2.3).

This section will discuss each solution and their viability.

5.2.1 Ksplice, kGraft, kpatch

Ksplice, kGraft and kpatch are all live kernel patching solutions. They allow to apply security patches to a running kernel without needing to reboot the system.

Ksplice
: Ksplice (Oracle) takes as input a unified diff and the original kernel source code, and it updates the running kernel in-memory. Ksplice determines what code within the kernel has been changed by the source code patch. Ksplice performs this analysis at the Executable and Linkable Format (ELF) object code layer, rather than at the C source code layer. The result is a loadable kernel module that performs the patching. [21]

kGraft
: The kGraft (SUSE) technology allows runtime patching of the Linux kernel by offering tools to create patch modules. The patch module is again a kernel module and fully relies on the in-kernel module loader to link the new code with the kernel. This again requires the original source code to compile against. [22]

kpatch
: Kpatch (Red Hat) offers a collection of tools which convert a source diff patch to a patch module, much like Ksplice. They work by compiling the kernel both with and without the source patch, comparing the binaries, and generating a patch module which includes new binary versions of the functions to be replaced. The patch module is a kernel module (.ko file) which includes the replacement functions and metadata about the original functions which can then be applied to the system. [23]

All of the solutions require access to the original source code to create unified diffs. Sadly, a lot of smartphone vendors do not release their kernel sources, even though this is in violation of the GPL [24]. When the source code is available, a device-specific kernel module can be made. Still, it remains a lot of work to create modules for each different device.

Some changes would have to be made to make these solutions work for the Android kernel. Moreover, Kpatch is only supported on Linux kernels with version 3.7 and above. On top of that, some smartphone manufacturers disable module loading or use kernel module signature verification (e.g. the AT&T Samsung Galaxy S4 [25]) so that third-party modules cannot be loaded. Even if it could be made to work, it is an inflexible approach.

5.2.2 Kprobes

Kprobes is a a debugging mechanism for the Linux kernel that can be used to dynamically hook any kernel routine and collect debugging and performance information non-disruptively. Almost any kernel code address can be trapped, specifying a handler routine to be invoked when the address is called [26].

To use Kprobes, a kernel module can be built that sets traps on the functions that need patching, while handling them with a patched version of the routine. The advantage is that full kernel sources are not needed, and an out-of-tree kernel module can be built. When building an out-of-tree LKM, only the kernel header files are needed instead of the full sources. There must also be a copy available of the matching module version information and version magic before building.

Module version information is stored in a file named `Module.symvers`, which is created during the kernel build. It lists all exported symbols from `vmlinux` and all modules. It also lists the CRC if `CONFIG_MODVERSIONS` is enabled [27]. Since an out-of-tree LKM build does not involve building the actual kernel, `Module.symvers` is generally not available. However, it can be (partially) rebuilt if there is access to the kernel symbols (section 5.2.3.1) or the system's `zImage` [28]. Once this information is available for a certain device, it can be reused on other devices with the same kernel.

The kernel version magic (vermagic) is also needed. When compiling the kernel from source, the version magic information may be found in the `include/generated/utsrelease.h` file. It contains parameters that are specific to the system, and is added to every kernel module. When loading

a kernel module onto a system, a check is performed to see if the LKM's vermagic matches the system's. This is an extra measure to prevent incompatibilities. More information can also be found in section 6.2.1.3.

The module would have to be compiled for every different device, but the approach has less dependencies since no full sources are needed. Support for Kprobes can be enabled at compile-time with the `CONFIG_KPROBES` kernel configuration variable. This means that Kprobes is not necessarily available on every device, which makes this approach once again unreliable.

5.2.3 Expatting

After observing the kernel patching solutions in the previous sections, two disadvantages stand out: often kernel sources are not at hand, and the created patches (in the form of loadable kernel modules) would be specific to the device. A universal solution is needed that works in either situation.

Another way of getting ring 0 access other than through a kernel module, is by exploiting a security vulnerability. These can be the exact same vulnerabilities as the ones that have to be patched later on. The approach of exploiting followed by patching will be referred to as *expatting*. This is a newly proposed term and is not used outside out of this report.

Through means of a kernel vulnerability, it may be possible to gain read/write access to kernel memory in some circumstances. The vulnerable kernel functions or variables (symbols) can then be patched by overwriting the associated memory with new object code. The addresses or locations of these symbols are different for each kernel and are therefore device-dependent. Consequently, these addresses have to be determined at runtime.

5.2.3.1 Kernel symbols

User space applications (and also the kernel) do not use symbol names like `OpenFile()`. Variable or function names are known by an address such as `0x34cf8000`, where they can be accessed in memory.

Address space layout randomization (ASLR) may make the exact location of where symbols can be found unpredictable. In order to prevent an attacker from reliably jumping to a particular exploited function in memory, ASLR

can randomly rearrange the positions of key data areas of a program, including the base of the executable and the positions of the stack, heap, and libraries, in a process's address space. In March 2014, support for address space randomization for the Linux kernel itself, which randomizes where the kernel code is placed at boot time, was merged into the kernel mainline of version 3.14 [29].

Android has had ASLR for user space applications since version 4.0. At this moment, with the release of version 4.4.4, ASLR is not being applied on kernel level, and symbol addresses are still being determined at compile-time [30].

At runtime, the addresses of kernel symbols can sometimes be found in the `/proc/kallsyms` (or `/proc/ksyms`) file. The output of `/proc/kallsyms` can be seen in Table 1. However, this file may be unavailable depending on the `CONFIG_KALLSYMS` kernel configuration variable. The `System.map` file also holds the kernel symbols, but is contained on the `/boot` partition, which is likely inaccessible.

Moreover, a new sysctl[1] `kptr_restrict` was added to the kernel source tree by Dan Rosenberg, which may hide the addresses of kernel symbols. In his commit, the formatting string to print the kernel symbols changed from `%p %c %s` to `%pK %c %s`. Quoting Rosenberg, the `%pK` format specifier is designed to hide exposed kernel pointers, specifically via `/proc` interfaces. Exposing these pointers provides an easy target for kernel write vulnerabilities, since they reveal the locations of writable structures containing easily triggerable function pointers. The behavior of `%pK` depends on the `kptr_restrict` sysctl [31]:

- `kptr_restrict` = 0: no deviation from the standard `%p` behavior occurs and addresses are visible by everyone.
- `kptr_restrict` = 1: only root can view the kernel symbols, while others see the kernel pointers printed as zeros.
- `kptr_restrict` = 2: kernel pointers using `%pK` are printed as zeros regardless of privileges (except in kernel space). This is the standard in Android.

Another way to get the symbol locations is through any kind of memory disclosure or memory device and subsequently searching for the symbol table. Getting read/write access to the kernel memory will be addressed in the next section.

[1] sysctl: an interface that is used to modify kernel parameters at runtime.

Table 1 Example output of unrestricted and restricted `/proc/kallsyms`

Unrestricted: `kptr_restrict` = 0	Restricted: `kptr_restrict` = 2
`c0008000 T stext`	`00000000 T stext`
`c0008000 T _text`	`00000000 T _text`
`c000804c t __create_page_tables`	`00000000 t __create_page_tables`
`c0008100 t __turn_mmu_on_loc`	`00000000 t __turn_mmu_on_loc`
`c000810c T secondary_startup`	`00000000 T secondary_startup`
`c0008148 T __secondary_switched`	`00000000 T __secondary_switched`
`c0008154 t __secondary_data`	`00000000 t __secondary_data`
`c0008160 t __enable_mmu`	`00000000 t __enable_mmu`
`c0008180 t __vet_atags`	`00000000 t __vet_atags`
`c00081c0 T asm_do_IRQ`	`00000000 T asm_do_IRQ`

On Samsung Galaxy S2, S3, Note 2 and some other smartphones, a character device file `/dev/exynos-mem` was available which gave access to all physical memory, and was read/write for all users. A public root exploit for this vulnerability searched for the format string `%pK %c %s` in memory and replaced `%pK` by `%p` to force the display of kernel symbol pointers [32].

5.2.3.2 Read/write access to kernel memory

Several device files exist that may lend themselves for kernel memory reading and writing.

The character device file `/dev/mem` is an image of the main memory. It may be used to examine (and even patch) the system. Byte addresses are interpreted as physical memory addresses [33]. If the kernel configuration variable `CONFIG_STRICT_DEVMEM` is set to enabled, user space access to `/dev/mem` will be limited to memory mapped peripherals [34]. Moreover, some smartphone vendors have patched the source code to completely remove the `/dev/mem` device file.

The file `/dev/kmem` is the same as `/dev/mem`, except that the kernel virtual memory rather than the physical memory is accessed. It is possible that the kernel has been configured with `CONFIG_DEVKMEM` disabled, in which case this device will not be accessible.

When `CONFIG_PROC_KCORE` is enabled, the system will have a `/proc/kcore` device file which corresponds to the system's physical memory. It returns data in the core dump file format. It does not provide an interface to write to kernel memory, and can only be used for reading memory.

In case none of these character devices are available, it is still possible to resort to a kernel vulnerability that may leak kernel memory (e.g. CVE-2013-6282 [35]).

5.2.3.3 Kernel exploiting and patching

To exploit and patch kernel vulnerabilities, techniques of rootkits can be used. Most rootkits traditionally perform system call hooking by swapping out function pointers in the system call table [36]. Not coincidentally, this same technique can be used for replacing vulnerable functions with a patched version.

Once there is write access to kernel memory, privilege escalation can be accomplished using the common technique of overwriting and triggering a kernel function pointer with the address of a payload in userspace. A root shell can then be spawned to fix framework vulnerabilities from.

Despite an effort to make all possible function pointers in the kernel read-only, certain design patterns still leave ample opportunity for exploitation. For example, by overwriting a function pointer within the `ptmx_fops` struct (which is not in read-only memory) associated with `/dev/ptmx`, it is possible to subsequently trigger the pointer with a call to `fsync` [37].

To achieve privilege escalation, the struct `cred` is the basic unit of "credentials" that the kernel uses to keep track of what permissions a process has. What user it is running as, what groups it is in, etc. The syscall `prepare_kernel_cred` will allocate and return such a struct with full privileges, intended for use by in-kernel daemons. Using `commit_cred`, the provided credentials can then be applied to the current process, thereby giving full permissions [38].

To determine the addresses of `commit_creds`, `prepare_kernel_cred` and `ptmx_fops`, these symbols have to be resolved. This process was described in section 5.2.3.1.

A few kernel exploiting frameworks exist that may make the exploiting step easier. Most notably *spender*'s (Brad Spengler) Enlightenment framework [39], and the Android rooting tools by *hiikezoe* (Hiroyuki Ikezoe) and *fi01*.

Patching the kernel can be done the same way as exploiting: swapping out the function pointer in the system call table, or by overwriting the vulnerable routine in-memory.

5.2.3.4 Kernel protections

In some cases, additional security measures have been taken that make the exploiting step tougher.

In Android 4.3, Security-Enhanced Linux (SELinux) was introduced to enforce Mandatory Access Control (MAC) over all processes [40]. SELinux can operate in one of two global modes: permissive mode, in which permission denials are logged but not enforced, and enforcing mode, in which permission denials are both logged and enforced. In Android 4.3, SELinux was fully permissive. In Android 4.4, SELinux was set to enforcing mode for several root processes: `installd`, `netd`, `vold` and `zygote`. All other processes, including all regular Android applications, remain in permissive mode to allow further evaluation of SELinux.

Some initiatives have looked into Android kernel hardening with grsecurity, such as AniDroid-Hardened [41]. Grsecurity consist of a set of patches for the Linux kernel that offer role-based access control, can frustrate and log exploit attempts, restrict access to certain syscalls, and hide information from `/proc`. Its PaX component can also provide ASLR for both user and kernel space, advanced bounds checking, memory erasure on `free`, and preventing execution of writable memory [42].

On x86 systems, there is a write protection bit that can be applied to kernel memory pages. Attempting to overwrite these read-only marked pages will result in a kernel oops[2].

However, most Android devices are based on the ARM architecture and do not enforce read-only kernel page permissions. On the other hand, one point of interest is that ARM CPUs utilize a data cache and instruction cache for performance benefits. Modifying code in-place may cause the instruction cache to become incoherent with the actual instructions in the memory. The solution is to flush the instruction cache whenever a modification to kernel code is made, which is accomplished by a call to the kernel routine `flush_icache_range` [36].

The NX bit, which stands for No-eXecute, is a technology used in CPUs to segregate areas of memory for use by either storage of processor instructions or for storage of data. This feature is in use on both x86 (as the XD-bit) and ARM version 6 and onwards (as the XN-bit) [43]. On Android, when the NX bit is supported, the syscall `vmalloc_exec` can be used to get a memory block with full access, or to adjust the access. For processors or operating systems that do not support the NX bit, execution of code is implicitly allowed [44].

[2]Kernel oops: a problem that arises in the kernel. In case one leads to a kernel crash, the term “kernel panic” is used.

5.3 Boot hooking

Since the proposed expatting techniques fix vulnerabilities in-memory, the patches will not persist after a system reboot. In this section, several strategies will be discussed on how to apply patches on boot, leaving the base system as intact as possible. Exploits and patches should be saved locally on the device so that there is no need to download them on every boot.

There are a few obstacles that have to be overcome, and each approach has its own advantages and disadvantages.

5.3.1 Obstacle: dm-verity

An example of an obstacle is device-mapper-verity (dm-verity). Android 4.4 supports verified boot through the optional dm-verity kernel feature. Device-mapper is a Linux kernel framework that provides a generic way to implement virtual block devices. It is used to implement volume management (LVM), full-disk encryption (dm-crypt), and in this case: transparent block device integrity checking (dm-verity). [45, 46]

The dm-verity feature looks at the block device, the underlying storage layer of the file system, and determines if it matches its expected configuration. It does this using a cryptographic hash tree. For every block (typically 4k), there is a SHA256 hash. This way, dm-verity may help to protect against persistent rootkits that can hold onto root privileges and compromise devices.

On Android, dm-verity is applied to the `/system` partition, which makes it impossible to make persistent system changes. Since it is no longer possible to write to e.g. the `/system/bin/app_process` binary, it becomes harder to hook into the runtime or onto the system boot.

5.3.2 Approach: broadcast receiver

Android applications can hook into the `BOOT_COMPLETED` broadcast. It is a non-ordered broadcast, meaning that it is sent to applications asynchronously in an undefined order. This broadcast is sent right after the system has started, and can be caught by implementing a `BroadcastReceiver` in the Android application. The Expat MDM agent can then proceed by patching the vulnerabilities. Since other applications can also hook into this broadcast, a race condition exists in which a malicious application can misuse a vulnerability right before the Expat MDM agent gets to action.

The current Rekey implementation, as discussed in section 5.1.1, also listens to the `BOOT_COMPLETED` broadcast, and is consequently vulnerable for this race condition.

5.3.3 Approach: modifying init files

A key component of the Android bootup sequence is the `init` program, which initializes elements of the Android system. It is different from regular Linux distributions which usually use some combination of `/etc/inittab` and SysV init levels [47].

The `init` process examines two files, `init.rc` and `init.$device.rc`, and executes the commands it finds in them. The first file is used for generic initialization instructions, while the latter is intended to provide device-specific initialization instructions.

Modifying the `init.rc` files is not straight-forward. They are part of the ramdisk and not the system partition. This means that at boot, the `init.rc` file stored in the ramdisk will be used, and whatever changes are made to `init.rc` will not be reflected as the file is overwritten. Making changes to the `init.rc` file requires unpacking the boot image, unpacking the ramdisk, editing the `init.rc` file, repacking the ramdisk, and repacking the boot image.

However, many manufacturers and ROM makers call their own init scripts from `init.rc`. These may execute scripts that are found in the `/etc/init.d` or `/etc/rc.d` directory. These init script may lend themselves for boot hooking, as commands can be appended to them. There is no guarantee that own init scripts are actually used, which makes this approach unreliable.

5.3.4 Approach: modifying app_process binary

The earlier discussed `init.rc` file starts the Zygote daemon via the `/system/bin/app_process` binary. As explained in section 4.1.1, Zygote is responsible for starting and managing application processes. Since Zygote preloads the Android application framework, it is a good hooking point. The Zygote process runs as root, and can therefore also be used to launch other boot hooks from.

The Xposed framework uses the `app_process` binary to hook into the runtime and override or extend framework functionality. For this, the original source code has to be modified and recompiled with the necessary changes. The Zygote daemon could for instance always give priority to the `BOOT_COMPLETED` broadcast receiver of the Expat MDM agent, which solves the race condition in section 5.3.2.

The downside is that the implementation of the `app_process` binary may change over time, and different versions for each Android version and architecture have to be compiled (but not for every different device).

To modify files on the `/system` partition, the partition has to be remounted with read-write rights. Furthermore, this approach cannot be used when dm-verity is enabled, as the binary is contained in the `/system/bin` directory. Verification of the `/system` partition will fail in this case.

5.3.5 Overview

A combination of the three boot hooking techniques could be used to add a boot hook that is both safe (no race conditions) and reliable. Table 2 gives an overview of the up- and downsides of the discussed approaches.

On devices with dm-verity enabled, init scripts could still offer a way to hook onto the system boot. If dm-verity is disabled, the most reliable way would be to modify the `app_process` binary. At all times, a broadcast receiver should be used that hooks onto the `BOOT_COMPLETED` broadcast. In case the init script or the Zygote binary failed to call the boot hook, there is always a backup solution. The broadcast receiver can also pick up on this failure (by checking if the other boot hook was executed) and subsequently try a different hooking strategy or init file to hook into.

Table 2 Boot hook strategy comparison

Strategy	Pros	Cons
init script	+ cross-platform	− init.rc overwritten on boot − init scripts not necessarily available
app_process binary	+ always in the same place	− dm-verity − architecture specific
broadcast receiver	+ cleanest	− allows race condition

6 Expat MDM

In general for MDM solutions, there is a central server that issues commands and receives status messages. The mobile device has an agent application installed that listens and acts on the received commands. The agent application can for instance enforce policy settings and report events.

When an MDM solution is capable of using the aforementioned exploit techniques to apply patches, it will be referred to as an expat(-capable) MDM solution in this section.

The structure of a typical Expat MDM setup is shown in Figure 4. Just like in a regular MDM setup, the MDM server and MDM agent can be found, but get complemented by extra functionality for expatting. The agent reports system information to the server and receives a set of patches that should be applied, along with the needed exploits to do so.

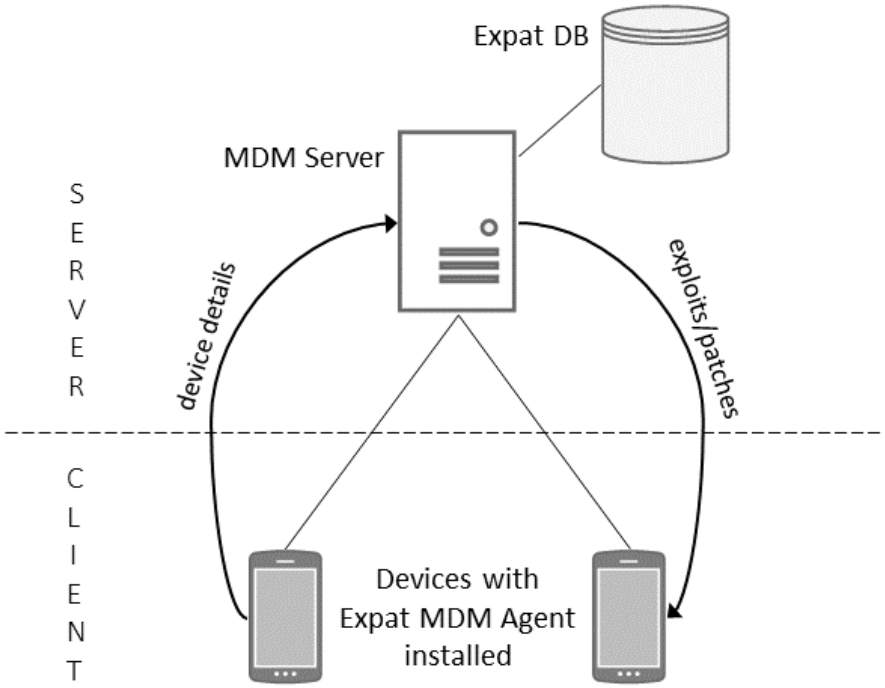

Figure 4: Agent-server Expat MDM setup

Details will be given on both components and what the device life cycle of an Expat MDM solution in the enterprise would look like, concluding with a proof of concept implementation of these ideas.

6.1 Server component

The Expat MDM server component receives status messages and system information that is submitted by the agent application. The set of patches that should be applied can then be determined for that device, along with the exploits that are likely to work.

The database that powers the Expat MDM server, as found in Figure 4, contains details about exploits and patches, such as:

- The exploit or patch type: e.g. code execution, information disclosure.
- The exploit or patch target: either the runtime or the kernel.
- Affected versions: version of the application framework or kernel.
- Affected architectures: in case of a kernel exploit, it may be specific to a certain system architecture, e.g. ARM.
- A link to the exploit or patch as compiled native binary object.

The business logic contained in the server component uses this data to determine which exploits and patches are suitable for the device that it is currently connected to. Priority should be given to exploits that are likely to succeed and unlikely to cause kernel panics. For example, some exploits may require a race condition that only occurs under certain circumstances. Successful exploitation may depend on the hardware or the compile-time kernel settings. Moreover, an exploit may overwrite parts of kernel memory while it is in use by another process, leading to undefined behavior or a crash. These are situations that should be avoided altogether by choosing correct priorities for available exploits. It may be acceptable to allow exploits that can lead to crashes by listening to kernel messages, and only disabling them if they actually cause problems (section 6.2.2).

6.2 Agent component

The agent component is an Android application that is installed onto the user's device and communicates with the server. It gathers all kinds of system information that are of interest to the expatting process, and submits this to the MDM server.

This information only has to be gathered once for the system. In case when ROM updates are pushed, the information gathering process should be done over. A caching mechanism should be in place that remembers this information, while saving the downloaded patches and exploits to a local directory. The patches can then be applied on boot without having to contact the MDM server first.

The proof of concept Expat MDM agent implementation, addressed in section 6.4, also provides code for the server communication and information gathering that is dealt with in the following sections.

6.2.1 Information gathering

The MDM agent will gather system information such as the kernel version and which runtime is in use. Note that there are no additional application permissions needed for the agent to discover this information, or any of the information in the following sections.

Determining if security measures such as SELinux, grsecurity or LKM signing are in use, may also be of interest, but is currently out of the scope of this paper.

6.2.1.1 Miscellaneous information

The most important relevant information can be easily queried using the built-in Java system property functions. Information that can be gathered through the `System.getProperty` function includes, but is not limited to: the Linux kernel version, the Android OS version, the architecture, and the device make and model (see Table 3).

Table 3 Android system properties

Property	Description	Example
android.release	The Android OS version	4.2.1
android.sdk	The SDK version of the framework	17
os.arch	The system architecture	armv7l
os.version	The Linux kernel version	2.6.32
build.brand	The consumer-visible brand	JIAYU
build.manufacturer	The device manufacturer	JYT
build.model	The device model	JY-G5

6.2.1.2 Determining the runtime

Knowing which runtime is in use is important for determining how framework vulnerabilities should be patched. The process of patching framework vulnerabilities is different under Dalvik than it is under ART, as they are fixed by hooking into the runtime.

Reading the system property `java.vm.name` currently returns "Dalvik" regardless of which runtime is in use. It is possible that this behavior will change once ART is no longer in its infancy.

As a workaround, the class `android.os.SystemProperties` provides an interface similar to `System.getProperty`, which reveals a property called `persist.sys.dalvik.vm.lib`. It returns `libdvm.so` for the Dalvik VM or `libart(d).so` for ART, and is currently the most reliable way of determining the used runtime. The class is not being exported as part of the public SDK, but can still be accessed using Java Reflection.

6.2.1.3 Linux vermagic string

If the kernel supports module loading, a Linux kernel module (LKM) may be loaded to gain access to kernel memory. The downside is that LKMs are system specific.

As briefly mentioned in section 5.2.2, one of the complications in building an out-of-tree kernel module is that the vermagic string contained in the kernel object file must match the system's version magic.

The vermagic string contained in a Linux kernel module may look as follows:

```
3.4.5 SMP preempt mod_unload ARMv7
```

This string means that the kernel was compiled for ARM devices against the codebase of version 3.4.5 with support for Symmetric Multi Processing (SMP), preemptive multitasking (PREEMPT), and module unloading (mod_unload).

The command `uname -a` may inform on version information of which the output will look as follows:

```
Linux localhost 3.4.5 #1 SMP PREEMPT Thu Oct 31 16:13:14 CST
2013 armv7l GNU/Linux
```

However, this string is incomplete (e.g. mod_unload is not supplied) and does not report the exact vermagic as the LKM would expect it to find.

The `/system/lib/modules` directory on Android devices usually contains kernel modules, and is also readable by Android applications. Since the vermagic string can be found inside the kernel object files (.ko), it suffices to scan the directory for files ending in this extension, and to locate the vermagic string in one of them.

6.2.1.4 Status of dm-verity

Kernels that are compiled with dm-verity will not allow changes to the `/system` directory. This complicates boot hooking in a reliable way (section 5.3).

Among the steps to enable the verified boot process, developers are required to add a `verify` flag to the device's `fstab` file in order to enable block integrity verification for the system partition.

A line in the `fstab` file would look like as follows to support dm-verity:

```
/dev/block/platform/msm_sdcc.1/by-name/system /system ext4
ro, barrier=1 wait,verify
```

Reading the `fstab` file and checking for the `verify` flag could thus be a way to detect whether dm-verity is enabled.

For Android 4.2.2 and earlier, the device-specific `vold.fstab` configuration file defines mappings from sysfs devices to filesystem mount points. For Android releases 4.3 and later, the various `fstab` files used by `init`, `vold` and `recovery` were unified in the `/fstab.$device` file [48]. The following locations could be searched for readable `fstab` files:

- `/fstab.$device`
- `/fstab.$device.rc`
- `/etc/vold.conf`
- `/etc/vold.fstab`
- `/etc/fstab`

Besides locating the `fstab` file, the `veritysetup` binary may be found in the `PATH`. The `veritysetup` command is used to configure dm-verity managed device-mapper mappings. The availability of this command could indicate that dm-verity is enabled.

6.2.2 Expatting process

The exploits and patches that the MDM agent receives from the server are in a binary format that is specific to the architecture. These binary objects can either be in the form of a shared libray or native executable. The objects are downloaded and saved to the application's `files` directory.

In case the object is a shared library, it should contain JNI bindings to be able to talk to the Android Java application. Using a shared object has as downside that JNI exported functions should be defined beforehand in the Java application. This can be circumvented by using a wrapper library with predefined JNI exports that loads other shared libraries using `dlopen`. A major downside is that if a shared object causes an error, it will cause the whole Android application to crash.

Using a native executable may therefore be a better choice than a shared library. As an extra step, the read/execute bits for the binary should be set with the `chmod` utility. The binary can then be executed using the `Runtime.getRuntime().exec()` function. In this case, the process is forked off from the main Android application. This provides an extra layer of resilience against errors, and will not cause the whole Android application to crash if the forked process crashes.

Since applications cannot write outside their folder except for the SD card, and the SD card is mounted with the `noexec` flag, only the application's `files` directory should be used to save native executables.

When exploiting is successful, a list of kernel symbols can be extracted from memory. This list should be saved, as it can speed up subsequent expatting after a reboot. It is also of interest for use on other devices with the same kernel.

Patches should be applied in a fault tolerant way. If exploits or patches are poorly programmed, they make cause kernel oopses and even kernel panics. For this reason, the procfile `/proc/kmsg` should be closely monitored, which is where kernel events will show up. In case of a kernel panic, messages from the previous boot can be found in `/proc/last_kmsg`. When a kernel panic is detected, the exploit or patch that led to the crash can be disabled.

6.3 Expat device life cycle

This section describes the life cycle of a device if it would be used in an Expat MDM environment. The focus is on use within the enterprise, although the same concepts can be applied on any MDM solution that exists outside a work environment.

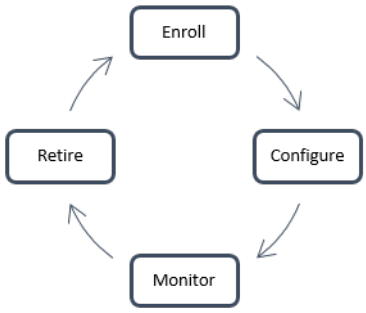

Figure 5: MDM device life cycle

The following device states can be distinguished when considering the life cycle of an Expat MDM device:

Enroll
An employee would like to use an own device for company purposes. The user chooses to agree with the enrollment and monitoring procedures. Enrollment steps are taken to allow the device onto the company network. This includes examining the smartphone if it meets all requirements. E.g. the device is not rooted and contains no malware.

Configure
The Expat MDM agent is installed onto the device. It enrolls itself with the MDM server, gathers device details, and sends these to the server. Initial exploiting and patching occurs, and a boot hooking strategy is determined.

Monitor
The device is being monitored for new threats, either by periodically checking with the MDM server or by pushing new updates to the device. Policy updates and security patches are applied as they come in.

Retire
The device is taken out of the MDM monitoring cycle. The agent is removed and any possible boot hooks or system modifications are reverted. The device can now be repurposed for use outside of the enterprise environment.

In regard to the Evard machine life cycle [49], an additional *Unknown* state could be added to the Expat MDM life cycle. For instance, having a rooted device can have implications on the integrity and workings of the MDM agent and its monitoring cycle. Rooted devices cause a certain entropy (e.g.

unreliable status information) that results in an *Unknown* device state. If the device is rooted even before the enrollment process, there is always additional entropy that has to be taken into account.

When the user decides to perform a manual firmware update, or when the manufacturer pushes an OTA update, the device goes back to the *Configure* state. Device details are gathered anew after which the correct exploiting, patching, and boot hooking strategies are determined, as they may have changed after the update.

6.4 Proof of concept

The expatting techniques introduced in this paper were consolidated into a proof of concept Expat MDM solution. The proof of concept consists of a server and agent component, and demonstrates how an Expat MDM solution would be used in real-world situations. The agent application is able to download exploits and patches from a central MDM server and apply them.

The server component for this proof of concept was written in NodeJS. It uses the Express web application framework to serve API requests. The Expat DB is an in-memory SQLite database that is accessed from the main app through the Sequelize ORM as extra abstraction layer.

The MDM agent is an Android application that gathers system information as described in section 6.2.1.

The POC contains a sample exploit and patch for CVE-2013-6282. It is compiled as a shared object from C code with JNI bindings. The exploit description, as originally reported by Catalin Marinas [35]:

> The (1) get_user and (2) put_user API functions in the Linux kernel before 3.5.5 on the v6k and v7 ARM platforms do not validate certain addresses, which allows attackers to read or modify the contents of arbitrary kernel memory locations via a crafted application, as exploited in the wild against Android devices in October and November 2013.

The demo code can be found in an online code repository (Appendix A).

7 Practical evaluation

The discussed techniques can be combined to create a device-independent vulnerability exploit-patching framework. There is no need to load a device-specific kernel module.

During the creation of a patch, assembly code is often used to perform in-memory code replacement. This means that although the patch can work regardless of the kernel compile-time symbols, the code may still have to be adapted for other architectures. Most Android devices are running on ARM, but this may change in the future.

The low-level character of the exploits and patches may cause kernel panics when offsets in memory are not carefully chosen. There is a need for fault tolerance: patches should be applied very carefully in order not to disturb the normal functioning of the system. The patches should contain verification routines to check if the patch has been applied correctly.

The in-memory patching nature of expatting is by default non-permanent, and if things go wrong, a reboot is enough to revert to a clean system. However, when using boot hooks, persistent changes can be made to the system to increase the reliability of expatting.

Some kernel vulnerabilities are race conditions that may take a while to gain kernel-level access. If the proposed techniques were to be applied on boot, this may make the startup time significantly longer.

8 Ethical considerations

Some of the proposed techniques can also be found in malicious software such as rootkits that have the intention to gain unwanted privileged access to devices. As always "with great power comes great responsibility" applies: a malicious Android application could obtain the same elevated privileges as the MDM agent without needing additional user intervention (through means of an exploit). That application could then misuse the privileges (e.g. destroy or steal data) whereas the Expat MDM agent can patch these exact vulnerabilities that led to the exploit.

In a BYOD setting, employees would first have to sign a consent form that states their awareness of the implications of having the MDM agent installed onto their devices. The techniques proposed in this paper make it possible to obtain privileged access (and thus also access to private data). These same techniques shall not be used for anything other than is needed to protect the device from the predefined or emerging vulnerabilities/threats.

An Android application that exploits/patches security vulnerabilities such as the proposed Expat MDM agent is not necessarily an enterprise solution, but can also be offered to individuals. In this case, the developer/vendor of the application should clearly state the earlier concerns. It is then up to the end user to trust the vendor to not abuse the obtained privileges.

Lastly, an application that implements the proposed expatting techniques can possibly not be released onto the Google Play store. X-Ray[3], a vulnerability scanner for Android, was disallowed from the Play store for not complying with the content policy.

The Expat MDM agent could be classified under the *Dangerous Products* or *System Interference* section of the Google Play content policy[4]. Especially the following items cause concern:

- Don't transmit or link to viruses, worms, defects, trojan horses, malware, or any other items that may introduce security vulnerabilities to or harm user devices, apps, or personal data.

  → The Expat MDM agent retrieves exploits and patches from a centralized server, which makes it incompatible with the Google Play content policy.

[3] http://www.xray.io
[4] https://play.google.com/about/developer-content-policy

– An app downloaded from Google Play may not modify, replace or update its own APK binary code using any method other than Google Play's update mechanism.

  → The exploits and patches that are received from the central server are binary object files that will be executed. It is unclear whether this would be an actual problem.

– An app downloaded from Google Play (or its components or derivative elements) must not make changes to the user's device outside of the app without the user's knowledge and consent.

  → As long as the user consents to the expatting process, this condition is complied with.

The agent can still be pushed to devices in a BYOD environment that uses an in-house application store. There are also alternative application markets that may not impose the same restrictions and allow the Expat MDM agent without further concerns.

9 Conclusion

Runtime and kernel hooking techniques (section 5) can be used to patch vulnerabilities in Android devices. Hooking solutions for the Dalvik runtime already exist (section 5.1). Several approaches are discussed to hook vulnerable functions in the kernel (section 5.2).

The proposed expatting techniques for the kernel can leverage weaknesses in the system to gain privileged access to the device and subsequently patch vulnerabilities in-memory (section 5.2). Hooking of vulnerable functions is possible by resolving kernel symbols at runtime (section 5.2.3.1). The operating system can then be patched from user space by using vulnerabilities or character device files that lend access to kernel memory (section 5.2.3.2). This device-independent method no longer requires the explicit need for the original kernel sources, or the loading of kernel modules. Because vulnerabilities are patched in-memory, patches have to be applied on each system boot (section 5.3).

Thanks to expatting, the vendor can be cut out, and the responsibility of bringing out patches can be transferred to the MDM solution (section 6). It offers a way to bring outdated Android devices up-to-date with the latest security fixes. The MDM agent gathers system details (section 6.2.1) that can be used by the MDM server to determine which patches should be applied on the device (section 6.1). An overview is given on what the life cycle of a device would look like as part of an expat-capable MDM solution (section 6.3).

There are a few difficulties that have to be overcome such as possible security measures (section 5.2.3.4), reliable boot hooking (section 5.3.5) and resilience against kernel panics (section 6.2.2). By carefully creating the exploits and patches, the latter can in most cases be overcome (section 7).

The implementation of the Expat MDM application (section 6.4) proves that the concept of expatting is feasible: it uses the same security vulnerability to exploit and subsequently patch the system. It also offers a basis for the creation of MDMs that employ the same expatting techniques.

10 Future work

This paper was mostly focused around kernel vulnerability patching. Since ART is poised to become the new default Android runtime, vulnerability research and hooking techniques to bring out patches for ART will be needed. Other kernel exploiting techniques are out there and may also be interesting to add to the ones mentioned.

Furthermore, researchers will have to observe how the use of security measures on Android, such as SELinux, dm-verity and ASLR, will affect expatting in the future.

Lastly, no database of security vulnerability exploits and patches currently exists for kernel nor framework vulnerabilties. If expatting would be used in the real world, such a database is needed.

11 References

[1] TechCrunch and Natasha Lomas. Android Still Growing Market Share By Winning First Time Smartphone Users. http://goo.gl/FJKHC6.

[2] Joost Kremers. Security Evaluation of Mobile Device Management Solutions. Master's thesis, Radboud Universiteit Nijmegen, 2014.

[3] Terrence Cosgrove, Rob Smith, Chris Silva, Bryan Taylor, John Girard, and Monica Basso. Magic Quadrant for Enterprise Mobility Management Suites. *Gartner G00211101, April*, 2014. https://info.mobileiron.com/gartner-magic-quadrant-2014-content.html.

[4] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. PatchDroid: scalable third-party security patches for Android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013. http://www.mulliner.org/collin/academic/publications/patchdroid.pdf.

[5] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, volume 10, pages 1–6, 2010. http://appanalysis.org/tdroid10.pdf.

[6] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013. http://www.nicta.com.au/pub?doc=6865.

[7] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011. http://www.seclab.tuwien.ac.at/papers/egele-ndss11.pdf.

[8] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. PREC: practical root exploit containment for android devices. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 187–198. ACM, 2014. http://dance.csc.ncsu.edu/papers/codespy14.pdf.

[9] Keunwoo Rhee, Woongryul Jeon, and Dongho Won. Security Requirements of a Mobile Device Management System. *International Journal of Security & Its Applications*, 6(2), 2012. http://www.sersc.org/journals/IJSIA/vol6_no2_2012/49.pdf.

[10] Keunwoo Rhee, Dongho Won, Sang-Woon Jang, Sooyoung Chae, and Sangwoo Park. Threat modeling of a mobile device management system for secure smart work. *Electronic Commerce Research*, 13(3):243–256, 2013.

[11] Dan Bornstein. Dalvik VM internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008. https://sites.google.com/site/io/dalvik-vm-internals/.

[12] Dan Bornstein. Android Developers Blog: Dalvik JIT. http://android-developers.blogspot.nl/2010/05/dalvik-jit.html.

[13] Bas Alberts and Massimiliano Oldani. Beating up on Android. http://www.infiltratecon.net/infiltrate/archives/Android_Attacks.pdf.

[14] Patrick Brady. Anatomy & physiology of an android. In *Google I/O Developer Conference*, 2008. http://androidteam.googlecode.com/files/Anatomy-Physiology-of-an-Android.pdf.

[15] Wikipedia. Protection ring. http://en.wikipedia.org/wiki/Protection_ring.

[16] Jools Whitehorn. Android malware gives itself root access. http://www.techradar.com/news/phone-and-communications/mobile-phones/android-malware-gives-itself-root-access-1062294.

[17] Wikipedia. Android rooting. http://en.wikipedia.org/wiki/Android_rooting.

[18] Rovo89. XDA-Developers: Xposed - ROM modding without modifying APKs. http://forum.xda-developers.com/xposed/framework-xposed-rom-modding-modifying-t1574401.

[19] Jeff Forristal. Android: One root to own them all. *Black Hat USA*, 2013. https://media.blackhat.com/us-13/US-13-Forristal-Android-One-Root-to-Own-Them-All-Slides.pdf.

[20] Rovo89. XDA-Developers: The ART of patience. http://forum.xda-developers.com/showpost.php?p=49979752.

[21] Wikipedia. Ksplice. http://en.wikipedia.org/wiki/Ksplice.

[22] Wikipedia. kGraft. http://en.wikipedia.org/wiki/KGraft.

[23] Red Hat. kpatch: dynamic kernel patching. https://github.com/dynup/kpatch.

[24] Wikipedia. Linux kernel: Legal aspects. http://en.wikipedia.org/wiki/Linux_kernel.

[25] jeboo. XDA-Developers: BypassLKM: bypass module signature verification on TW 4.3. http://forum.xda-developers.com/showthread.php?t=2578566.

[26] Jim Keniston and Prasanna S. Panchamukhi. Kernel debugging with kprobes. https://www.kernel.org/doc/Documentation/kprobes.txt.

[27] Kernel.org. Building External Modules. https://www.kernel.org/doc/Documentation/kbuild/modules.txt.

[28] Mike Hommey. Building a Linux kernel module without the exact kernel headers. http://glandium.org/blog/?p=2664.

[29] Wikipedia. Address space layout randomization. http://en.wikipedia.org/wiki/Address_space_layout_randomization.

[30] Jon Oberheide. Exploit Mitigations in Android Jelly Bean 4.1. https://www.duosecurity.com/blog/exploit-mitigations-in-android-jelly-bean-4-1.

[31] Dan Rosenberg. kptr_restrict for hiding kernel pointers from unprivileged users. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=455cd5ab305c90ffc422dd2e0fb634730942b257.

[32] MWR Labs. Extra modules for the Mercury Android Security Assessment Framework. https://github.com/mwrlabs/mercury-modules.

[33] About.com. Linux / Unix Command: mem. http://linux.about.com/library/cmd/blcmdl4_mem.htm.

[34] Android Git. Kernel hacking Kconfig.debug. https://android.googlesource.com/kernel/common/+/android-3.0/arch/unicore32/Kconfig.debug.

[35] MITRE. CVE-2013-6282. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282.

[36] Michael Coppola. Suterusu rootkit: Inline kernel function hooking on x86 and ARM. http://www.poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/.

[37] Michael Coppola. CSAW CTF 2013 kernel exploitation challenge. http://www.poppopret.org/2013/11/20/csaw-ctf-2013-kernel-exploitation-challenge/.

[38] Nelson Elhage. Much ado about NULL: Exploiting a kernel NULL dereference. https://blogs.oracle.com/ksplice/entry/much_ado_about_null_exploiting1.

[39] Brad Spengler. Enlightenment - Linux Null PTR Dereference Exploit Framework. http://www.exploit-db.com/exploits/9627/.

[40] Android Source. Validating Security-Enhanced Linux in Android. http://source.android.com/devices/tech/security/se-linux.html.

[41] Novic_dev. XDA-Developers: AniDroid-hardened kernel. http://forum.xda-developers.com/nexus-s/development/kernel-anidroid-hardened-t1525257.

[42] Richard Carback. Understanding Linux kernel vulnerabilities. http://www.csee.umbc.edu/courses/undergraduate/421/Spring12/02/slides/ULKV.pdf.

[43] Wikipedia. NX bit. http://en.wikipedia.org/wiki/NX_bit.

[44] ITB CompuPhase. The SMALL booklet: implementor's guide. 2005. http://www.doomworld.com/eternity/engine/smallguide.pdf.

[45] Android Source. Dm-verity on boot. https://source.android.com/devices/tech/security/dm-verity.html.

[46] Nikolay Elenkov. Verified boot with dm-verity. http://nelenkov.blogspot.be/2014/05/using-kitkat-verified-boot.html.

[47] eLinux. Android Booting. http://elinux.org/Android_Booting.

[48] Android Source. Device specific configuration. https://source.android.com/devices/tech/storage/config.html.

[49] Rémy Evard. An Analysis of UNIX System Configuration. In *LISA*, volume 97, pages 179–194, 1997. https://www.usenix.org/legacy/publications/library/proceedings/lisa97/full_papers/20.evard/20_html/main.html.

Appendix

A Code repository

The proof of concept code for the Expat MDM solution is too long to include in the appendix, and can therefore be found on GitHub at http://github.com/c3c/ExpatMDM. It uses CVE-2013-6282 by *fi01* for both the exploiting and patching.

Table 4 Repository structure

Directory	Description
android/	The Expat MDM Android agent PoC
nodejs/	The Expat MDM server PoC code in NodeJS
poc/	Contains the native exploit/patch code in C with JNI

B Acronyms

ADB Android Debug Bridge

AOT Ahead-of-time (compilation)

API Application Programming Interface

APK Android Package

ARM Acorn RISC Machine

ART Android RunTime

ASLR Address Space Layout Randomization

BYOD Bring Your Own Device

DEX Dalvik Executable

JAR Java Archive

JIT Just-in-time (compilation)

LKM Linux Kernel Module

LVM Logical Volume Management

MAC Mandatory Access Control

MDM Mobile Device Management

ORM Object-relational Mapping

POC Proof of Concept

ROM Read-only Memory. In this context, firmware for the mobile device.