PatchDroid: Scalable Third-Party Security Patches for Android Devices

Collin Mulliner
Northeastern University
crm@ccs.neu.edu

Jon Oberheide
DuoSecurity
jono@dueosecurity.com

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

ABSTRACT

Android is currently the largest mobile platform with around 750 million devices worldwide. Unfortunately, more than 30% of all devices contain publicly known security vulnerabilities and, in practice, cannot be updated through normal mechanisms since they are not longer supported by the manufacturer and mobile operator. This failure of traditional patch distribution systems has resulted in the creation of a large population of vulnerable mobile devices.

In this paper, we present PatchDroid, a system to distribute and apply third-party security patches for Android. Our system is designed for device-independent patch creation, and uses in-memory patching techniques to address vulnerabilities in both native and managed code. We created a fully usable prototype of PatchDroid, including a number of patches for well-known vulnerabilities in Android devices. We evaluated our system on different devices from multiple manufacturers and show that we can effectively patch security vulnerabilities on Android devices without impacting performance or usability. Therefore, PatchDroid represents a realistic path towards dramatically reducing the number of exploitable Android devices in the wild.

1. INTRODUCTION

Google's Android operating system has become the largest mobile device platform, with over 750 million devices worldwide and about 1.5 million new activated devices every day[1]. Android is popular with users since there are many applications – about 800,000 at the time of writing – available for the platform. Mobile devices take advantage of the fact that the main operating system and applications framework are maintained by Google and made available for third-party customization and distribution. Because of these reasons, almost all major mobile device manufacturers produce Android-based devices.

Google constantly publishes enhancements and bugfixes for the Android platform. To quickly distribute updates, since it's early days, Android provided an over-the-air (OTA) update mechanism through which devices can be updated over the network without being attached to a computer. Every time a security vulnerability is found and fixed, Google can push an update to all Android devices. In theory, this mechanism allows all Android phones to receive security-relevant patches in a timely fashion.

Unfortunately, this statement is only true for devices sold directly by Google and that are currently supported. Devices produced by other manufacturers are not updated by this mechanism; furthermore, updates are somewhat regulated by mobile network operators – e.g., some operators require review of new firmware versions prior to deployment. Therefore, the responsibility for distributing security updates lies primarily in the hands of the individual device manufacturers. Some manufacturers are better than others when it comes to updates and especially security fixes. But most manufacturers only support a specific product for a limited time frame, often on the order of 1–2 years. After that time, manufacturers do not provide any further updates even though those older devices remain in widespread use.

As a result, a large percentage of Android phones in use run an outdated version of the Android operating system. According to Google, in May 2013[2] 38% of all Android devices were running Android 2.3 (Gingerbread), Gingerbread is known to contain many severe security vulnerabilities, including multiple local privilege escalation, arbitrary code execution via drive-by-downloads in the browser, and permission leaks where unprivileged application can perform operations that normally require special permissions. Publicly released exploits exist for most of these vulnerabilities. In most cases, these exploits were originally written for jailbreaking purposes. However, they have also proven useful in writing malware, for instance to escalate privileges on infected devices [17].

Malware is far from the only problem with unpatched, unsupported Android devices. For example, corporate espionage and targeted attacks have become a major issue in the area of mobile devices. Companies and governments need to be able to protect their devices against attacks, but are currently left at the mercy of device manufacturers and mobile network operators for receiving security updates in a timely fashion.

Our goal in this work is to bridge the gap between official publication of security patches by Google and distribution of those patches to poorly-supported devices. In particular,

[1]http://gigaom.com/2013/04/16/android-on-track-for-1b-total-activations-later-this-year-google-chairman-says/

[2]http://developer.android.com/about/dashboards/

ACSAC '13 Dec. 9-13, 2013, New Orleans, LA, USA
Copyright 2013 ACM 978-1-4503-2015-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2523649.2523679.

we propose a new system called PatchDroid that provides safe and scalable third-party patching of security vulnerabilities in unsupported Android devices. PatchDroid does not rely on help from either the manufacturer or operator since it does not rely on recompilation of source code released by the manufacturer. Instead, PatchDroid is based on dynamic, in-memory patching of running processes. Through this mechanism, security analysts can write a patch for a specific vulnerability once, and deploy the patch using PatchDroid on all Android devices that are vulnerable to the specific bug. PatchDroid uses dynamic binary instrumentation in order to inject patches into running processes on Android. As part of this work, we have developed techniques to patch bugs in native processes as well as in managed code that is executed in the Dalvik VM.

In addition to vulnerability remediation, PatchDroid also includes an attack detection subsystem to recognize exploitation attempts against the fixed vulnerabilities. If an attack attempt is detected, the attack mitigation subsystem produces both on-device user notifications as well as remote reporting.

We have evaluated PatchDroid on multiple devices from different hardware manufacturers running a wide variety of Android versions. Our evaluation demonstrates that we can effectively patch known security vulnerabilities on all tested devices. The operational state of the devices were, in all cases, unaffected by PatchDroid, and the process did not induce any noticeable performance overhead. Finally, we recruited beta testers to run PatchDroid on their devices, and attempted to run exploits against those devices. The results show that PatchDroid prevented a successful exploitation in all cases.

This paper makes the following contributions.

- We present PatchDroid, a system for safely distributing and applying third-party patches for security vulnerabilities in the Android operating system. The approach allows analysts to construct patches that can be applied across a large set of mobile devices and OS versions for both managed Dalvik bytecode as well as native code.

- We present techniques for performing in-memory patching, which improves the safety of our system by guarding against persistent modifications to critical system files that could lead to "bricked" devices. In-memory patching also sidesteps issues surrounding modification of signed system partitions.

- We evaluated PatchDroid over a number of Android devices from different device manufacturers running various legacy Android versions (2.0 to 4.1). Our evaluation demonstrates that PatchDroid is safe, does not incur noticeable system overhead, and effectively prevents exploitation of security vulnerabilities in Android devices.

The rest of this paper is organized as follows. Section 2 provides a short overview of Android and the Android vulnerability landscape. In Section 3, we present the motivation for this work. Section 4 presents the design of PatchDroid. In Section 5, we present our patching methodology. In Section 6, we provide details of our implementation. Section 7 presents our evaluation of PatchDroid's reliability, performance, and effectiveness in preventing exploitation. In Section 8, we discuss related work, and Section 9 concludes the paper.

2. BACKGROUND

In this section, we provide a brief overview of the Android OS and the vulnerabilities that exist at each layer of the system.

2.1 The Android OS

The Android OS can be conceptually broken down into four layers. The lowest layer is the Linux kernel, which provides basic operating system services such as memory management, process separation, and device drivers. The next layer is the Android runtime, system daemons, and support libraries. The Android runtime is implemented using the Dalvik Virtual Machine. The Dalvik VM as well as the system daemons and support libraries are executed as native code. The last two layers are the Android application framework and the actual applications. Both are implemented in managed code and are executed by the Android runtime using the Dalvik VM.

2.2 Vulnerabilities in Android

Vulnerabilities exist in every layer of the Android OS from the kernel to the framework and applications. Mainly, vulnerabilities in native code that allow arbitrary code execution are of interest since they provide the basis for attacks. Most interesting are vulnerabilities in privileged programs since they can be used for privilege escalation attacks. In the scope of this paper, we only looked at user space vulnerabilities. A general list of known Android vulnerabilities of different nature is available here [8].

For the presented work, we picked five known vulnerabilities. Four very well-known and *widely-exploited* privilege escalation vulnerabilities that exist in native code, and one vulnerability in the SMS stack that is implemented in managed code, and executed in the Dalvik VM. The vulnerabilities are of different kinds and, thus, provide a good general overview of bugs. Missing range checks for array access in vold lead to the GingerBreak [25] – this bug is also abused by known malware [17]. Vold further contains a buffer overflow that lead to the zergRush [26] exploit. Both zygote and adbd contained a bug that is based on not checking return values from setuid(). This lead to the zimperlich [16] and RangeAgainstTheCage [15] exploits, respectively.

Capability leaks such as those found by the woodpecker tool [10] are a good example of bugs in Dalvik code that can be actually exploited for malicious intent. For instance, one such capability leak allows any application to spoof an incoming SMS message, including setting an arbitrary number as the sender. This bug can be used for SMiShing (SMS-Phishing) attacks [24]. A proof-of-concept exploit for this bug has been released [6].

2.3 Vulnerability Uncertainty

Google does not force manufacturers to adhere to any particular versioning schema for their Android devices; as such, manufacturers can use any Android version. In general, most manufacturers use the most recent version in order to benefit from new features to better compete in the market. However, this does not mean that each manufacturer will apply security patches to their version of the Android source after adapting it for their specific needs. The result of this is that Google's version of a particular Android version might be patched against a specific vulnerability, while the manufacturer's version of the same Android release might still be vulnerable. Therefore, it is not necessarily straightforward to determine which Android device is affected by which bug by the release version alone. This situation has led to the creation of X-Ray [9], a vulnerability scanner for Android.

Tests conducted using X-Ray [19] show that two devices, running non-Google versions of Android 2.3.6 and 3.2, are both still vulnerable to the GingerBreak bug even though Google officially patched [25] it in Android version 2.3.4 and 3.0.1.

3. MOTIVATION

Our motivation in this work is fixing security vulnerabilities in Android devices that are no longer supported by the manufacturer and mobile network operator. Additionally, we want to provide the ability to roll out third-party security fixes, which can greatly reduce the population of vulnerable Android devices in the wild.

Our main goal is to fix vulnerabilities without relying on device manufacturers to provide source code or source-level patches for security issues. Access to source code such as that published by AOSP[3] can be leveraged to gain insights on how known vulnerabilities were fixed by Google.

A further goal is scalability. We wish our approach to be scalable across different device manufacturers and device models. To that end, we do not want to create flashable operating system images mainly because not all devices support flashing new operating system images that are not signed by the manufacturer. Creating complete operating system images without access to the source code of the manufacturer is possible by combining AOSP with the binary-only parts from the manufacturer. However, this has the downside of being unable to fix bugs in manufacturer-specific portions of the OS. That is, it would require significant testing, and further would lose all manufacturer-specific software that is installed on the phone.

3.1 Challenges

We identified a number of challenges in the course of this work.

No source code and version uncertainty. We do not have access to the source code of vulnerable applications and libraries. Even in the optimal case where a vulnerability is in a software component that is contained in the open source part of Android, we do not necessarily know which version of the code was used on a specific device. Furthermore, the manufacturer might have changed the original source code to add or modify functionality. Finally, even if one had access to the source code for an affected device, it would be a significant undertaking to compile the affected code.

Issues with static binary patching. Static binary patching has the advantage that one does not have to collect source code and set up a build environment that can accommodate a large number of different devices. However, binary patching would need to address issues such as different processor versions (i.e., ARMv5 vs. ARMv7), different optimizations, and compile flags. Additionally, ARM code can be distributed as either full 32-bit instructions, or as Thumb code which uses 16-bit instructions. Therefore, static binary patching would require to patch each binary individually.

[3]AOSP refers to the Android Open Source Project, which publishes the open source component of the Android OS.

Writing to system partitions. For both approaches outlined above, there remain deployment issues. Since we would like to avoid distributing full system images, we would instead have to replace individual files on the filesystem. However, all system binaries and support libraries on Android devices are stored on read-only file systems. Simply mounting the file systems to be writable does not solve the problem. If the replaced file does not work correctly on the given device, the device might be rendered unusable. The device might suffer from unreproducible crashes, reboots, or perhaps stop booting at all. Additionally, some manufacturers cryptographically sign system partitions and check the signatures of these partitions during boot using the manufacturer-specific boot loader. Modifying or adding binaries on system partitions, therefore, would be error-prone and, in some cases, virtually impossible.

Scalability vs. testing. Due to the large number of different manufacturers and devices, scalability becomes an issue. The main issue we are concerned about is testing. Two main problems arise here. First,devices are based on different code bases. In order to scale patch development, it has to be decoupled from the devices original source. Second, it is practically impossible to test each patch on every affected device. This problem becomes worse if this includes differences in Android OS versions and regional customizations. Scalable patch development and testing are an important issue in order to fix vulnerabilities on a large-scale.

3.2 Goals

In light of the challenges above, we adopted the following design goals for our system.

Patch development scalability. A patch should only need to be written once and should work on all affected Android versions and devices.

Reliability. Patching a vulnerability must not affect device reliability. For instance, the device must not experience random crashes or be rendered unusable.

Native code vs. Dalvik code. Vulnerabilities exist both in native binaries and in software written in Java and executed in the Dalvik VM. Our system must be able to patch vulnerabilities in both worlds.

Scalable and fast deployment. Patches generated by our system must be efficient to distribute over the network, simple to install, and easy to reverse should the need arise. Patches should be applicable to all possible known device configurations, as well as previously unknown configurations to a reasonable degree. Patch deployment should explicitly allow for blacklisting specific patch and device combinations if they crash on a given device.

Attack detection. Our system should be able to detect exploitation attempts performed against patched vulnerabilities. It should be possible to report detected attacks to both the user as well as a central entity.

4. DESIGN

PatchDroid, our system for distributing and applying third-party patches to Android devices, is composed of a number of components that reside both in the cloud as well as on end-user devices. The end goal of these components is to en-

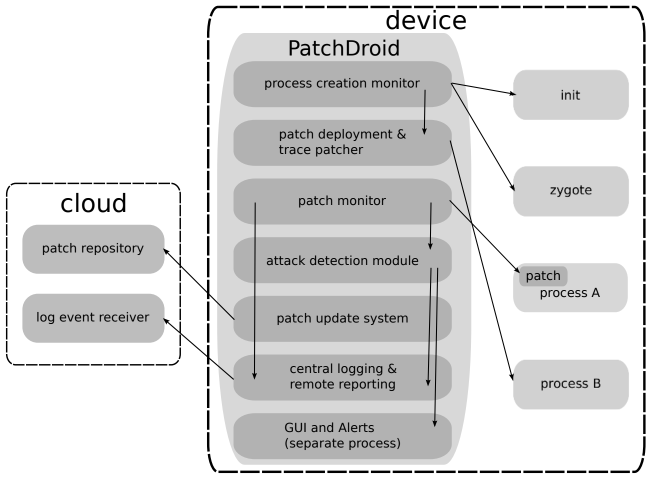

Figure 1: PatchDroid architectural overview.

able safe, dynamic, in-memory patching of vulnerable programs and libraries, as in-memory patching avoids all of the challenges described in Section 3. A visual representation of this architecture is shown in Figure 1.

In the following, we present a high-level overview of each of these components and the information flows between them. We defer discussion of the implementation details of each of these components to Section 6.

4.1 Device Components

On end-user devices, PatchDroid introduces a number of distinct components that cooperate in order to apply patches and monitor the system.

Patch injector. The *patch injector* is responsible for deploying patches into running processes. This component also verifies that the patch is required for the specific Android version before it is deployed. Furthermore, it checks if a particular patch is known to have previously caused crashes on a device in a given configuration. The information is supplied by the *patch monitor*.

Patch monitor. The *patch monitor* is responsible for monitoring the execution of code injected as part of a patch to determine if it is causing instability. Additionally, it collects log messages and attack warnings issued by the patch.

Process creation monitor. The *process creation monitor* interposes on process creation to determine if patch injection needs to be activated for the given process. Monitoring is carried out on `init` for system services and on `zygote` for Dalvik-based processes.

Attack detector. The *attack detector* interoperates closely with the patch monitor. Its primary purpose is to analyze attack warning messages issued by a patch, and process the messages for reporting. If a new attack is detected, it notifies the user and the cloud service via the *alerter* and *reporter* components.

Patch updater. The *patch updater* periodically checks for new patches with the cloud service. If new patches are available, they are downloaded over a secure channel and their integrity is verified before they are put into the deployment pool.

Reporter. All components collect information, such as attack attempts, the catalog of installed patches, and patch stability telemetry. All such information is transmitted by the *reporter* to the cloud service to evaluate and improve the patch system.

Alerter. The *alerter* provides a user interface for some basic information about the patches being deployed. This component is a special part of the GUI that is activated to alert the user with the *attack detector* each time an exploitation attempt is detected.

4.2 Cloud Components

In addition to the on-device components, two components comprise the cloud service portion of PatchDroid.

Patch repository. The *patch repository* provides access to a centralized store of available patches. On-device updaters query this component to determine whether patches are available for application on a device.

Log collector. The *log collector* records device telemetry exported by the on-device reporter component, and performs analytics to discover patterns of patch instability or infection campaigns.

5. ANATOMY OF A PATCH

In this section, we present several low-level techniques used by PatchDroid for patching vulnerabilities in both native code as well as Dalvik bytecode for the Android platform.

5.1 Patching Native Code

PatchDroid applies patches for native code by replacing vulnerable functions with equivalent functions that do not contain the vulnerability. We denote these functions as *fixed* functions. Installing a fixed function is performed via inline hooking or by hooking the global offset table (GOT).

Our patch method for native code is based on shared library injection. Each patch is a self-contained shared library that is injected into the target process. Patches are loaded via the dynamic linker, and the patch code is executed through the library's `init` function that is called by the dynamic loader. The patch is activated in two steps. First, it sets up the communication link between the patch code and patch monitor, which is used for logging and error reporting. The second step is installing the actual patch; the details of this step vary depending on the type of patch. PatchDroid considers three possible scenarios when patching vulnerabilities, and thus employs three generic methods for applying patches that we describe in the following. Figure 2 depicts a graphical overview of these strategies.

5.1.1 Function Replacement

In the case where a vulnerability is localized to a single function, PatchDroid replaces the function with a fixed version of the same function. In the simplest case, one can acquire the source of the target application or library from the public AOSP repository and extract the target function.

5.1.2 Fixing via Function Proxy

Complex functions can be difficult to patch while ensuring that legitimate functionality has not been broken. In these cases, PatchDroid can elect to inject a function proxy that enforces a form of input sanitization to the vulnerable function. The proxy function wraps the original function, and inspects the function inputs and relevant global state to check for conditions that indicate an attack attempt. If a

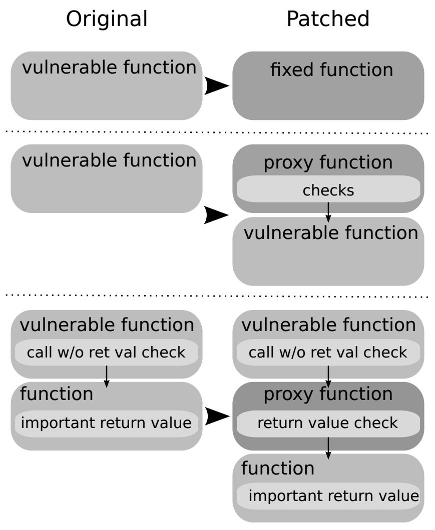

Figure 2: The three patching strategies employed by PatchDroid.

known malicious input state is determined to exist, the vulnerable function is not executed and an error is returned. Otherwise, the original function is executed on the input state.

5.1.3 Failed Return Value Checking

In addition to enforcing constraints on function input states, PatchDroid also allows for the enforcement of return value checks. In some cases, a vulnerability exists because a function neglects to check the return value from a function that it invokes. The strategy PatchDroid employs is to wrap the invoked function with a proxy that checks the return value on the caller's behalf before returning to the caller. This has the benefit of avoiding the complexity of replacing large, complicated functions.

5.2 Patching Dalvik Bytecode

PatchDroid conceptually adopts similar techniques to the case of native code when patching vulnerabilities in Dalvik bytecode. However, applying patches to Dalvik bytecode is not as straightforward as in the native case. Therefore, we developed a technique that allows us to replace arbitrary methods in Dalvik code with a native function call using JNI [21].

In contrast to native patches, which are bootstrapped using a shared library `init` function as described above, bytecode patches are bootstrapped through a one-time hook of a commonly called function such as epoll_wait() using inline hooking. Once that hook is executed, the actual patch is installed. The installation requires resolving a number of symbols in the Dalvik VM library (libdvm). The patching is implemented by replacing a Dalvik method with a native method using the DVM's JNI. The main steps in this process are (a) obtaining a class reference through `dvmFindLoadedClass`, which takes the fully-qualified class name as input; (b) obtaining a method reference via `dvmFindVirtualMethodHierByDescriptor`, which takes the method name and method signature as arguments; (c) and, replacing the Dalvik method with a native method via `dvmUseJNIBridge`, which takes the method reference and a function pointer as arguments.

6. IMPLEMENTATION

Our PatchDroid prototype implements each of the components described in Section 4. In the following, we describe the implementation details of the device-level components, our PatchDroid Android application for end users, and the patches themselves.

6.1 patchd: The Patch Daemon

The patch daemon, patchd, is the core of PatchDroid. It is launched at system startup and runs continuously in the background. Its main responsibilities are monitoring the system for new processes in order to apply any necessary patches prior to process execution, performing the actual patch application, monitoring the stability of patched processes, and logging attack attempts and unstable patches.

On startup, patchd collects and inspects the properties of the device it is running on via the Android properties API. The main system properties collected include CPU parameters such as the ARM core version; device and manufacturer information, including the platform and board name; and, the version of the Android OS.

After the system information is collected, patchd loads the meta data for all available patches. Patch meta data describes the name of the target process, the filename of the library that contains the patch, a descriptive name for logging purposes, the Android version that the patch is built for, and a set of flags to indicate specific options for the patch deployment process.

Once the patch database has been loaded, patchd performs an initial round of patch injection into already-running processes. In particular, patchd loops through the list of available patches and compares the process name from the metadata of every patch to the name of running processes to determine which patches should be deployed. This initial round of patch application utilizes the same rules and functionality as the runtime patch deployment system, which we describe in the following.

6.2 Patching Processes

patchd relies upon several core techniques to reliably and safely apply patches to running processes: process creation monitoring, patch deployment, and patch injection.

6.2.1 Process Creation Monitor

Monitoring the system for new processes on Android is implemented by tracing the init and zygote processes. init is traced to monitor for newly created processes that are the result of crashed system services such as zygote or vold. zygote is traced to monitor for startup of Dalvik-based processes. Tracing is implemented via the ptrace API. To reduce overhead, we use the `TRACE_FORK` and `TRACE_EXEC` functionality of ptrace. This feature of ptrace only pauses a traced process on calls to `fork`, `clone`, and `exec`, and avoids inducing performance overhead during execution phases that are not relevant for PatchDroid.

Once patchd has identified a newly created process, it attaches to it using ptrace and identifies the base executable or Dalvik class name by inspecting `/proc/$PID/cmdline`. The resulting name is used to search the patch database for a matching patch. In the case of a match, the patch and its metadata is forwarded to the patch deployment subsystem.

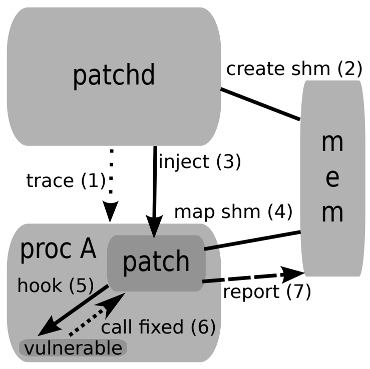

Figure 3: Life cycle of a patch. 1) The process is traced as it is created. 2) Shared memory is created. 3) The patch is injected, after that tracing is stopped. 4) The patch executes and maps the shared memory. 5) The patch inserts hooks. 6) The fixed function is called. 7) Patch reports back to patchd.

6.2.2 Patch Deployment

Patch deployment is the term we use to describe the overall process of patching a process. It is composed of two steps. The first step ensures that the patch is suitable for application to the target process, while the second step is the actual patch injection. Figure 3 shows the life cycle of a patch from injection to execution of the fixed function.

Patch constraints.

Before a patch is injected into a process, several checks are performed to ensure system stability. First, the Android version running on the device is compared with the target Android version in the patch metadata. The version check is coarse-grained and is generally restricted to the major Android version. The minor version is only compared in rare cases where it is clearly known that a certain version is not vulnerable to a specific bug. The issue of version uncertainty was discussed in detail in Section 2.

A second check that is performed considers the impact of failed patches. If a particular patch was injected before but was observed to cause crashes, it will not be loaded again. This feature is intended to ensure device stability, and is essential for our ability to test patches in the field.

Finally, target mismatches encompass failures to initialize and load a patch correctly. For instance, a missing symbol could cause this check to fail.

Once it has been determined that a specific patch should be deployed in a target process, patchd will inject the shared library implementing the patch into the target process.

Patch injection.

Patch injection is based on library injection via the ptrace API. Injection proceeds in three steps. First, patchd writes a loader stub to the stack of the target process consisting of code along with data such as the full path to the shared library that should be loaded. In the second step, the loader stub is executed, which invokes `mprotect` to mark the stack as executable.[4] The loader stub loads the patch shared library using `dlopen` and restores the register state after `dlopen` returns. In the final step, the dynamic linker transfers execution to the injected library by calling the library's `init` function.

[4]The patch library will later revert the memory protection to its previous state.

Performing reliable library injection entailed overcoming several non-trivial challenges.

Process blocked on syscall. If a process (e.g., zygote) is blocked on a system call, the library injection has to be postponed until the syscall is completed. Some syscalls cannot be restarted, and some processes simply crash due to internal state corruption shortly after library injection. In our tests, we determined that most of the time, processes are blocked on I/O polling functions such as `select` or `poll`. Also common, however, are sleep operations such as `nanosleep`.

There are two solutions for this problem. One approach is to wait until the process completes the syscall by itself. However, this is rather inconvenient as it is undesirable to monitor and track blocked processes.

Instead, a better solution is to stimulate the process to coerce it to return from the syscall. For zygote, this can be done by forcing the creation of a new DVM instance, for instance, by launching an application or service. In PatchDroid, this feature is implemented by part of the PatchDroid user-level application that is presented later in this section.

Patching newly started processes. The ptrace API allows patchd to reliably track and interpose on process creation. However, new processes need to execute until a certain point before the patch library can be injected into the process without crashing it. There are multiple reasons for this restriction. For instance, the dynamic linker must be given the chance to load the executable and any necessary libraries, and during this time, it is not safe to inject the patch library. Additionally, patchd cannot inject the library while the process is executing a syscall, since some syscalls cannot be not restarted. Therefore, finding the proper point to inject the patch library is not straightforward.

patchd uses a set of heuristics to determine when it is safe to inject the patch library. For instance, in the case of zygote, we determined that the process is safe to patch when it changes its name to zygote from its parent process (app_process). patchd covers this scenario by tracing the process until it changes its name. The tracing is performed using `TRACE_SYSCALL` to check for the process name change after it returns from syscall invocations. We note that even though this operation seems expensive, our evaluation shows that it is not. The actual monitoring is carried out through our general patch deployment subsystem that constantly checks the process name against the list of patch targets. Once a process name matches a patch target, the process is patched after which the heavyweight `TRACE_SYSCALL` tracing is stopped. Since Dalvik processes are created by zygote forking off a new process, PatchDroid is able to handle every Dalvik process in this way.

For native processes, patchd waits until a specific syscall such as `select` is executed that indicates that the process has reached a stable phase of execution. After the syscall returns, patchd pauses the process and injects the patch library.

Handling statically-linked code.

Patching statically linked code cannot be accomplished using library injection. To handle this case, patchd implements a feature we term *trace patching.* The trace patcher extends the process creation monitor and the patch injection code. Trace patching uses `TRACE_SYSCALL` tracing to interpose on syscall invocations in order to monitor for exploitation attempts. If an attempt is detected, the trace patcher simply kills the process.

While this is a relatively intrusive technique, virtually no processes on Android devices are statically-linked (i.e., to save space on the filesystem). To date, only one vulnerability affecting adbd has required the use of this technique, and in this case, tracing is only required for a short period during its initialization; afterwards, adbd is not known to be vulnerable and so tracing is discontinued.

6.3 Patch Monitoring and Attack Detection

The patch monitor is the subsystem responsible for monitoring patches running inside of a patched process. Communicating with the patch code is required for multiple purposes, most importantly for stability monitoring and attack detection.

Patch monitoring is implemented in our prototype using shared memory between the patched process and patchd. The shared memory region is created by patchd just before injecting the patch library into the target process. Once the patch's `init` function is executed, the patch code maps the shared memory region and has access to its content.

We elected to use shared memory as the IPC mechanism between the patched process and patchd for multiple reasons. Shared memory has very low overhead, and therefore, incurs a minimal performance impact. Using sockets would potentially require additional permissions due to the Android permission system. Since not all processes request this permission, sockets are unsuitable in this context.

On Linux, multiple possibilities exist for sharing memory between processes. Since the `shm` API is not supported on all Android devices, we use `mmap`-based shared memory.

The details of the main use cases for patch monitoring are discussed below. In addition, the shared memory segment is used by patchd to configure runtime parameters of a patch if necessary.

6.3.1 Patch Reliability

Reliability is one of the main goals of PatchDroid. In order to enforce reliability, we added the requirement for entry end exit counters in each patch. These counters are incremented every time the patch code is entered and exited. This allows patchd to ensure that the patch code is actually executed and to detect if the patch code caused a crash of the process. A crash inside the patch is indicated if enter $>$ exit. The counters are inspected every time a patched process dies and is restarted.

In addition, the patch can store a *failure code* to indicate failures such as a missing symbol. Missing symbols indicate that the patch and the process are not compatible. This information is recorded so that the patch will not be injected into the target process. This event is also forwarded to the backend reporting system.

6.3.2 Attack Detection

Attack detection is implemented via a trigger condition for each patch. The trigger condition evaluates to true in the case that an exploit attempt is performed against the patched vulnerability. The patch support code implements a reporting function that communicates with patchd using the shared memory channel. The attack attempt reporting incorporates a counter that is increased every time the trigger condition is satisfied and a timestamp of the last counter increment. In addition, the PID of the attacking process can be reported if it can be determined, for example through the metadata of a netlink socket.

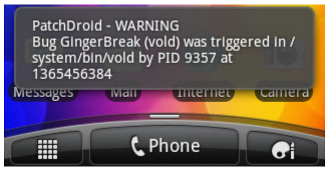
PatchDroid - WARNING
Bug GingerBreak (vold) was triggered in /system/bin/vold by PID 9357 at 1365456384

Figure 4: PatchDroid, running on a HTC WildFire, notifies the user about a detected exploitation attempt.

6.3.3 Log Facility

The log facility simply provides the ability for the patch code to log arbitrary information using the shared memory region. The log information is collected by patchd. The log facility is also used by the attack detection code to provide additional information about attack attempts.

6.4 The PatchDroid App

The PatchDroid application implements multiple features of PatchDroid that are required outside of patchd. For instance, the app is responsible for installing patchd and the patch libraries.

It also supplies the following features of PatchDroid: the HelperService, the Attack Notification UI, and the log file uploader. The log file uploader uploads the patchd log file to our web service for analysis.

6.4.1 HelperService

The *HelperService* is used during patch deployment. The HelperService is a Android service whose sole purpose is to wake up zygote by forcing it to start the HelperService by sending it a broadcast intent. In response, the service starts up and terminates immediately.

6.4.2 Attack Notification UI

When PatchDroid detects an exploitation attempt, it notifies the user through a broadcast receiver contained in the PatchDroid app. The patch daemon creates an intent that contains a message to be displayed to the user. The app receives the intent and displays the message to the user as shown in Figure 4.

6.5 Patches

We implemented proof-of-concept patches for four vulnerabilities: Zimperlich, GingerBreak, zergRush, and local SMS spoofing. Each patch is implemented as a self-contained shared library, this provides the capability to load the library from outside the `/system/lib` directory. Common code such as to lookup and hook functions and to access the shared memory segment to communicate with patchd is statically-linked into each patch.

As an example, we discuss the patch we developed to fix the setuid() vulnerability in zygote (the Zimperlich exploit). This patch consists of only the init and the fixed_setuid() functions. The init function maps the shared memory and inserts our fixed function to be called in place of setuid(). Figure 5 shows the code for our implementation of fixed_setuid(). It contains the stability tracking counters as well as the attack detection trigger.

```
 1    int fixed_setuid(uid_t uid)
 2    {
 3       track->c_enter++;
 4       int res = orig_setuid(uid);
 5       if (res == -1) { // possible attack
 6         pd_signal_attack(track);
 7         track->c_exit++;
 8         exit(0);
 9       }
10       track->c_exit++;
11       return res;
12   }
```

Figure 5: Fix for the setuid() vulnerability in zygote. The entry and exit counters are handled in Lines 3, 7, and 10. The original function is called on Line 4. Line 5 implements the check of the return value. Lines 6 and 8 handle the attack condition.

7. EVALUATION

In this section, we present an evaluation of our PatchDroid prototype. The goal of the evaluation is threefold: (a) to demonstrate that known vulnerabilities are effectively blocked, (b) to measure the performance overhead incurred by PatchDroid, and (c) to demonstrate that PatchDroid is stable and usable for end users.

7.1 Functional Evaluation

For the functional evaluation, we acquired multiple Android devices running different Android versions that contain a number of known vulnerabilities. The test devices and exploits we used for this evaluation were the *HTC Wildfire S* Android 2.3.3, vulnerable to GingerBreak and ZergRush; *Motorola FlipOut* Android 2.1, vulnerable to GingerBreak, ZergRush, and Zimperlich; (a) and, *HTC One V* Android 4.0.1, vulnerable to Dalvik-based SMS spoofing.

Our methodology for this evaluation was to: (a) exploit a vulnerability on a device without PatchDroid installed to demonstrate that the device is indeed vulnerable to a specific vulnerability; (b) install PatchDroid; (c) and, run the exploit to determine whether PatchDroid prevents successful exploitation. Some exploits determine and output if the target device is vulnerable or not.

Most of the privilege escalation exploits [25, 26] are packaged as a single binary that is copied to the device and is executed from an ADB shell. Other exploits, such as Zimperlich [16] come in the form of an Android (APK) that must be installed and run from the application launcher. For the local SMS spoofing vulnerability, we used an open source proof-of-concept [6] to trigger the vulnerability and to evaluate our patch.

For all device and exploit combinations, PatchDroid successfully patched the vulnerability, stopped the exploit, and notified the user of the attack attempt.

7.2 Performance Overhead

The goal of our performance evaluation was to determine the overhead that is imposed by running PatchDroid on an Android device. Benchmarking PatchDroid is not straightforward since it does not modify the system significantly.

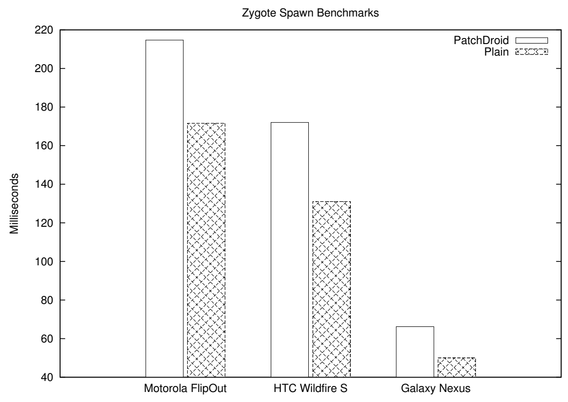

Figure 6: Overhead for starting new processes added through PatchDroid.

`patchd` traces `init` and `zygote` using `ptrace`, but only executes expensive operations during and shortly after process creation. The patches themselves incur a slight overhead as well, but only at the moment their code is executed. Hence, none of the available benchmarking tools were able to measure a statistically significant deviation from baseline performance resulting from `patchd`. Therefore, we designed our own microbenchmarking tool called *ZygoteBench*. ZygoteBench measures the time that it takes zygote to create a new process on an Android device.

ZygoteBench consists of two components, an application and a service marked to be run as a separate process. Every time the main application starts the service, zygote forks a new process for the service class. The service measures the time difference between issuing the start command and when the service class is actually executed. After the measurement, the service stops itself, killing the service process. For each measurement, the service is started ten times and the result is the mean of the ten measurements.

We ran ZygoteBench on three of our test devices before and after installing PatchDroid. Specifically, we ran our benchmarks on a Motorola FlipOut, an HTC Wildfire S, and a Samsung Galaxy Nexus. The Motorola FlipOut and HTC Wildfire had patches installed against vulnerabilities in vold and zygote. The Galaxy Nexus had one patch installed in system_server. Figure 6 shows that there is only a minimal and negligible overhead added through PatchDroid. On the Galaxy Nexus, we measured an overhead of 16 milliseconds on average for creating a new process and starting a service by zygote. We measured an average of 43 and 41 milliseconds overhead for the FlipOut and Wildfire respectively. These small increases in process creation time are not noticeable by the user in any way.

7.3 User Trials

To gain greater assurance in PatchDroid's stability and efficacy, we recruited a number of users to install and run PatchDroid on their devices. We then asked these users to run exploits against their devices while running PatchDroid.

Table 1 shows the devices for our evaluation. The table includes all devices that we tested PatchDroid on, including our own devices we used during development. The result of this study showed that 100% of exploits run against PatchDroid were successfully prevented.

#	Device	Android	Verified Bugs	External	Exploit Attempt
1	Motorola FlipOut	2.1	Zimperlich		X
	Sony Ericsson U20i	2.1	Zimperlich	X	X
	HTC Desire	2.2 (cm6)	?	X	
	Samsung GT-P1000	2.2.0	?	X	
	HTC Desire	2.2.2	?	X	
	HTC Wildfire s	2.3.3	GingerBreak + zergRush		X
	Nexus S (Samsung)	2.3.3	GingerBreak	X	X
	Sony Ericsson R800x	2.3.3	GingerBreak + zergRush	X	X
	HTC Droid Incredible	2.3.4	?	X	
10	Samsung SCH-I510	2.3.6	?	X	
	Galaxy Nexus (Samsung)	4.0.1	local sms spoofing		X
	HTC One V	4.0.4	local sms spoofing		X
	Lenovo P700i	4.0.4	?	X	
	Sony Ericsson Xperia Ultimate HD	4.0.4	?	X	

Table 1: Tested devices. Verified Bugs = vulnerable to bug, External = device not owned by us, Exploit attempt = exploit was run and PatchDroid prevented it.

7.4 The Master Key Bug and ReKey

While this paper was under review, the so-called Master Key bug [13] was disclosed. The bug can be used for privilege escalation by adding malicious functionality to APKs that are signed by a platform (manufacturer) key. We implemented a patch for the Master Key bug using our PatchDroid system.

Due to the critical nature of the bug, we decided to provide the patch to the general public. For this, we created the ReKey application. ReKey is a subset of PatchDroid restricted to its patch injection functionality. ReKey[5] is available in the Google Play Store and currently has about 12,000 active installs.

7.5 Discussion

patchd requires root privileges to be able to attach to processes such as zygote. Therefore, users can only install PatchDroid on already rooted devices. However, since PatchDroid mainly targets privilege escalation vulnerabilities, we could bootstrap PatchDroid by exploiting one of the vulnerabilities to gain root privileges and subsequently patching the vulnerability in order to secure the device.

The PatchDroid application registers itself to be run at device startup in order to launch patchd. This opens the possibility for race conditions against malicious software since there is no guarantee that PatchDroid is started first. However, with root privileges we could modify the system to ensure that PatchDroid starts before any other potentially malicious processes.

8. RELATED WORK

We are not the first to investigate dynamic runtime updating. The POLUS [7] system is based on patch library injection that is similar to our solution. However, POLUS requires access to the source code of the target application. Furthermore, the system is designed to keep software running continuously while updating it to new versions. Therefore, it implements a substantial amount of additional functionality to track and reset state. In contrast, our approach is more lightweight since we specially target fixing security problems and, thus, do not need to worry about issues such as changing data structures. Pin [12] performs instrumentation by taking control of the program just after it loads into memory. However, unlike the work we describe in this paper, Pin does not run on Android and does not focus on fixing vulnerabilities on legacy Android systems. Similarly, Dyninst [1] is a multi-platform runtime code-patching library. It provides an API to permit the insertion of code into a running program. Like Pin, Dyninst does not run on Android and does not focus on fixing vulnerabilities. Other efforts along these lines also either rely on source code availability, or changes to the tool chain (e.g., [5, 2, 14, 18, 11, 20, 3]). Our solution shares goals and techniques found in third-party patching and hotpatching systems for MS Windows [23].

Our *trace patching* method shares similarities with systrace [22]. Systrace enforces relatively simple policies on syscalls made by applications. In contrast, trace patching is focused on vulnerability mitigation, and is capable of enforcing complex constraints over program states due to its use of the ptrace API.

The ksplice [4] system provides a method for hot patching the Linux kernel by loading specialized kernel modules. Since Android uses the Linux kernel, ksplice could be adapted to implement Android kernel patching.

9. CONCLUSIONS

Vasts numbers of mobile devices in the field run outdated versions of their operating system and software stack. One notable example are mobile devices based on the Android platform. Almost 40% of all Android devices run software that is more then 24 months old and is no longer supported by the device manufacturer. Most of these devices contain severe security vulnerabilities that can be used for arbitrary code execution and privilege escalation.

In this work, we presented *PatchDroid*, a system to patch security vulnerabilities on legacy Android devices. PatchDroid uses dynamic instrumentation techniques to patch vulnerabilities in memory, and uses a patch distribution service so that patches only have to be created once and can be deployed on every device. Because patches are injected directly into processes, PatchDroid does not need to flash or modify system partitions or binaries, making it universally deployable even on tightly controlled devices.

We evaluated PatchDroid using devices from different manufacturers and, in addition, conducted user trials on a small group of users. Our evaluation shows that our method ef-

[5]http://play.google.com/store/apps/details?id=io.rekey.rekey

fectively fixes security vulnerabilities on legacy Android devices. It does not produce any noticeable performance overhead and is suitable to be used in the real world. Through the public release of ReKey, we gained further confidence in the effectiveness of our approach.

We believe that our method for fixing security vulnerabilities has broader application than Android-based mobile devices. Our system provides insights into how third-party distributed patching can be realized for general embedded devices that are no longer supported by a manufacturer.

Our current approach is restricted to patching userspace vulnerabilities within system frameworks or user applications. As future work, we plan to investigate extending our methods to kernel vulnerabilities.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under grant N000141310102, the National Science Foundation (NSF) under grant CNS-1116777, and Secure Business Austria.

Engin Kirda thanks Sy and Laurie Sternberg for their generous support. We would also like to thank everyone who helped by evaluating early versions of PatchDroid, with special thanks to Joshua J. Drake.

10. REFERENCES

[1] DynInst. `http://www.dyninst.org/`.

[2] Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. OPUS: Online Patches and Updates for Security. In *Proceedings of the USENIX Security Symposium* (2005).

[3] Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. OPUS: Online Patches and Updates for Security. In *In 14th USENIX Security Symposium* (2005), pp. 287–302.

[4] Arnold, J., and Kaashoek, M. F. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the ACM EuroSys Conference (EuroSys 2009)* (Nuremberg, Germany, March 2009).

[5] Bratus, S., Oakley, J., Ramaswamy, A., Smith, S. W., and Locasto, M. E. Katana: Towards Patching as a Runtime Part of the Compiler-Linker-Loader Toolchain. *International Journal of Secure Software Engineering (IJSSE) 1*, 3 (September 2010).

[6] Cannon, T. Android SMS Spoofer. `https://github.com/thomascannon/android-sms-spoof`, 2012.

[7] Chen, H., Yu, J., Chen, R., Zang, B., and chung Yew, P. Polus: A powerful live updating system. In *in Proc. of the 29th Intl Conf. on Software Engineering* (2007).

[8] CVE Details. Google : Android : Security Vulnerabilities. `http://cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html`, 2013.

[9] Duo Securtiy. X-Ray for Android. `http://xray.io`.

[10] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)* (February 2012).

[11] Hicks, M., Moore, J. T., and Nettles, S. Dynamic software updating. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (2001).

[12] Intel. Pin - A Dynamic Binary Instrumentation Tool. `http://software.intel.com/en-us/articles/pintool`.

[13] Jeff Forristal. Uncovering Android Master Key that makes 99% of devices vulnerable. `http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key/`, July 2013.

[14] Johnson, N. M., Caballero, J., Chen, K. Z., McCamant, S., Poosankam, P., Reynaud, D., and Song, D. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011).

[15] Kramer, S. Rage aginst the cage - adbd root exploit. `http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz`, 2010.

[16] Kramer, S. Zimperlich zygote root exploit. `http://c-skills.blogspot.com/2011/02/zimperlich-sources.html`, 2010.

[17] Lookout Inc. DroidDream. `http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/`, March 2011.

[18] Makris, K., and Bazzi, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (2009).

[19] Martin, J. X-Ray App Identifies Android Vulnerabilities But Doesn't Fix Them. `http://blogs.cio.com/smartphones/17286/x-ray-app-identifies-android-vulnerabilities-doesnt-fix-them`, 2012.

[20] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. Practical dynamic software updating for C. *SIGPLAN Notices 41*, 6 (June 2006).

[21] Oracle. Java Native Interface (JNI). `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html`.

[22] Provos, N. Improving host security with system call policies. In *In Proceedings of the 12th Usenix Security Symposium* (2002).

[23] Sotirov, A. Hotpatching and the Rise of Third-Party Patches. `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sotirov.pdf`, July 2006.

[24] T-Mobile. SMiShing and SMS Spam. `http://www.t-mobile.com/Company/PrivacyResources.aspx?tp=Abt_Tab_PhishingSMishing&tsp=Abt_Sub_IdentityTheft_SMiShing`, 2013.

[25] The Android Exploid Crew. CVE-2011-1823 - vold vulnerability "GingerBreak". `http://www.cvedetails.com/cve/CVE-2011-1823/`.

[26] The Revolutionary development team. CVE-2011-3874 - libsysutils rooting vulnerability "zergRush". `https://code.google.com/p/android/issues/detail?id=21681`, 2011.