Denial-of-App Attack: Inhibiting the Installation of Android Apps on Stock Phones

Steven Arzt[1], Stephan Huber[2], Siegfried Rasthofer[1], Eric Bodden[1 2]
CASED / EC SPRIDE
[1]Technische Universität Darmstadt [2]Fraunhofer SIT
[1]{firstname.lastname}@cased.de
[2]{firstname.lastname}@sit.fraunhofer.de

ABSTRACT

We describe a novel class of attacks called *denial-of-app* that allows adversaries to inhibit the future installation of attacker-selected applications on mobile phones. Adversaries can use such attacks to entrap users into installing attacker-preferred applications, for instance to generate additional revenue from advertisements on a competitive app market or to increase the rate of malware installation. Another possibility is to block anti-virus applications or security workarounds to complicate malware detection and removal.

We demonstrate such an attack that works on arbitrary unmodified stock Android phones. It is even possible to block many applications from a list predefined by the attacker instead of just a single app. Even more, we propose an attack for banning applications from Google Play Store regardless of the user's phone by exploiting similar vulnerabilities in the market's app vetting process. Unblocking an application blocked by our attack requires either root privileges or a complete device reset. The Android security team has confirmed and fixed the vulnerability in Android 4.4.3 (bug 13416059) and has given consent to this publication within a responsible-disclosure process. To the best of our knowledge, the attack applies to all versions prior to Android 4.4.3.

1. INTRODUCTION

Over the last decade, smartphones have become important personal and business devices. Many people rely on these devices to organize their appointments, read and send e-mails, manage personal and business contacts, or entertain themselves. Popular smartphone operating systems such as Android, iOS, or Windows Phone are built around a vibrant ecosystem that allows independent application developers to provide new software for the respective platform. This ecosystem is convenient for the user since almost every need for features is met, but also for the developers who can earn money by providing applications: either directly through selling paid applications, or through "free" applications in which they display advertisements. Especially the latter has become a big source of revenue, in some cases yielding several thousand dollars per day [11]. In total, the app market is expected to reach a volume of about 35 billion dollars in 2015 [22]. Still, like any other market, the smartphone application market requires fairness between offers and competitors: Customers (i.e., smartphone users) must be able to freely choose between offerings (i.e., applications). In this paper, we present a new class of attacks through which a developer can block applications from users' devices, effectively hindering competitors in distributing their products. For many use cases, several competing applications are available on the market. If a user cannot install one application, she will likely switch to the next one which offers a similar feature set. In the case of free, but advertisement-based apps, this means that the revenue for showing the ads goes to the developer of the second app, and not to the developer of the first one. As a consequence, being able to block applications can lead to increased financial revenue for the adversary in the scenario of competing application developers.

SPSM'14 November 07 2014, Scottsdale, AZ, USA
ACM 978-1-4503-3155-5/14/11 ...$15.00.
http://dx.doi.org/10.1145/2666620.2666621.

Even worse, current smartphone malware often disguises itself by forming a plugin to variants of otherwise benign, well-known applications [24]. With our attack, the attacker can entrap users to download such malware-infected app versions by inhibiting the installation of the respective original, benign app. Additionally, one could block the installation of anti-virus applications which would detect the malicious app.

The Google Play Store contains various security applications that check for certain well-known Android exploits, such as for the Master-Key Vulnerability [9], OpenSSL Heartbleed on Android [13], or the USSD-Exploit [20]. By using a *denial-of-app* attack, the installation of these security applications can be blocked, hindering the detection and prevention of further attacks on the device.

We present examples of the new *denial-of-app* class of attacks on Android which is the most popular smartphone operating system at the moment with a market share of more than 80% [12] and over 1 million apps in its official market alone, the Google Play Store. We have implemented a proof-of-concept exploit with which an arbitrary application can be blocked from being installed on a device. The exploit does not need any special privileges and runs on a stock Android phone without root access. To remove the application block imposed by the exploit, however, the user does need either root access and detailed system knowledge or must reset the complete phone to factory defaults, thereby losing all data on the phone if no remote backup of the data exists. In either case, the user needs to spend considerable effort to remove the block. Furthermore, as we show, the attack disguises itself as a normal installation problem, leading to many users staying unaware of a successful app block on their phones.

To the best of our knowledge, the exploit works on all Android versions. We have verified it on real devices running Android 2.3.6 and different Android 4.x versions (Android 4.1 - 4.4). Additionally we have verified it on emulated Android versions 2.2 to 4.4.2. Vulnerabilities that exist on a broad variety of Android versions are particularly critical since many devices are still running outdated versions of the operating system which are no longer supported by the device manufacturer and no longer receive security updates [2], which in turn means that they will stay vulnerable even if the problem is patched in upcoming Android versions. Furthermore, many smartphones run versions of the Android operating system that have been customized by the respective device manufacturer. In addition to the time required for Google to fix a security vulnerability, the device manufacturer must in such cases also adapt the fix into its own code tree and deploy a new build. This substantially enlarges the window of exposure or leads to fixes not being distributed at all [7]. In the context of our attacks, this has the consequence that large-scale market manipulations affecting lots of users will stay possible in the foreseeable future despite a fix being available now. We have reported all of our attacks to the Google security team and they have fixed the concrete vulnerabilities described in this paper[1]. This publication was submitted as part of the responsible-disclosure process.

The remainder of this paper is structured as follows. Section 2 gives important background information on the Android platform. Section 3 describes in detail the attack on the Android platform, whereas Section 4 gives examples on how the attack may be exploited in real-world scenarios. Countermeasures and recovery from the attack are discussed in Section 5 while Section 6 presents some related work known to literature and Section 7 concludes the paper.

2. BACKGROUND

Regardless of whether an Android application is installed from the official Play Store, a third-party app store or from a file on the SD card, the application is always contained in an APK file—which is just a renamed ZIP file—consisting of various files, most importantly a *dex* file and a *manifest* file. The *dex* file contains the application's executable code as Dalvik bytecode instructions. The *manifest* file describes important settings for the application such as the permissions it requests and the operating system version for which the application was built. It is the responsibility of the installer to check, among others, whether all files (APK, dex, manifest) are of the correct format, whether the target OS version in the manifest file is compatible with the actual OS version, and whether the application has already been blacklisted by Google as the OS vendor. Only if all these checks succeed, the app may be installed. Trusting the installer with this task is a reasonable design decision since the installer is a system component running under the sole user ID "system" that has write privileges to the directory in which Android stores applications in the internal storage. This concept prevents other applications from acting as installers on their own and makes the pre-deployed installer a central enforcement point.

[1]`https://android.googlesource.com/platform/frameworks/base/+/52af2ca919c068f1c9389fa4c979d2fe3105af40`

When the app has been verified successfully and the user has confirmed all of the privileges requested by the app, the installer copies the APK file to `/data/apk`. Afterwards, it creates a new Linux user account and group for the application. Recall that the Android platform enforces application sandboxing by running all apps in different instances of the Dalvik VM which in turn run under separate user accounts, one per application. Every application is assigned a private data directory at `/data/data` for storing assets such as configurations, caches, or user data. This directory is secured using Linux file-system permissions: only the application's individual user account is allowed to access it.

In current versions of the Android operating system, the account names under which applications run are composed of a prefix and an increasing number, e.g., "u0_a42". The name of the APK file on disk and the name of the application's private data directory are, however, equal to the app's package name as it is defined in the manifest file. The package name of an app can be seen as a UID, which has to be unique for each app. This has the consequence that no two applications with the same package name can be installed. This limitation can, however, not directly be exploited since the user can always uninstall any "blocking" application that may reserve the package name of an application she is trying to install[2].

3. DENIAL OF APP ATTACK

In this section, we describe our attack in more technical detail and explain the bug in Android's application installation system that makes our exploit possible (Section 3.1). The concrete exploit is constructed in Section 3.2.

3.1 App Verification Bug

Our attack exploits a bug in the installation sequence described above. It is at a very early point in time that the installer creates the user account, the group, and the private data directory for an application to be installed. At this time, not all checks on the APK file and its contents have been completed yet. If one ore more checks fail afterwards, these early actions need to be undone, as otherwise conflicting artifacts will be left in the internal storage. Having a spurious user and group ID in the system is not a big issue since the next application to be installed will simply get a higher number in the account and group name, e.g., ("u0_a43") instead of ("u0_a42"). The private data directory, on the other hand, receives the name of the application package as defined in the manifest file. If this directory is left over from a failed application installation (i.e., our exploit), it produces a conflict when another application with the same package name is attempted to be installed.

Recall that private data directories are owned by the respective application's account in the file system. This account is the only one with access to its contents. When a new application is to be installed, the installer cannot simply overwrite the directory, nor can it just transfer ownership, as for security reasons the installer account does not have root privileges either. Instead, the installer gives an error message. Depending on the user interface used for installing the new app, this is either reported in full, such as with

[2]Nevertheless, identifying the blocking application on its own is not a trivial task for the average user as the Android UI shows friendly names, not package names.

```
@Overwrite @Overwrite
public void killerMethod(){
  //do nothing
}
```

Listing 1: Code example which causes an installation failure of an Android application

the Android Debug Bridge (ADB)[3], or "installation failed" is given together with a numeric error code as in the Play Store app (e.g., see Figure 2). In either case, the attack cannot be distinguished from a normal installation failure unless one knows the attack, recognizes the error message and explicitly looks for installation leftovers, i.e., a private data directory without a corresponding application. Even more, since the application was not fully installed, it does not appear in Android's list of installed applications and thus cannot be uninstalled through the user interface. Uninstallation attempts on the command line such as `adb uninstall <packagename>` also fail with Android reporting that there was no application with the specified package name. Removing the leftovers to "unblock" the respective package name is therefore virtually impossible for the average user. Section 5 discusses this problem in more detail.

3.2 Exploit Construction

For staging the attack, one must build an APK file that makes the Android installer fail after creating the application's private data directory. This means that the manifest file must be valid, or otherwise the installer would abort too early. We therefore inject a semantic bug into the dex file containing the application code as the bytecode verifier *dexopt* (note that *dexopt* is used for bytecode verification and optimization) runs exactly between creating the directory and registering the application in the system. The purpose of the bytecode verifier is to check the application's code against the specification of the Dalvik VM. Mainly for performance reasons, but also to enforce semantic consistency, Dalvik enforces numerous constraints: Try/catch blocks, for instance, must be ordered by their starting line numbers and must not overlap, and no two annotations of the same type may exist for the same code object such as a method or parameter. To violate such a constraint, we use the Soot [21] bytecode optimization framework to read in an Android *dex* file, and write it out again (without applying any additional transformers to it). We patch Soot, however, such that while generating the output, it writes out annotations twice instead of just once. Listing 1 shows such an example. Installing the resulting APK file on an Android phone fails with a *dexopt* failure as expected. Using `adb install` this is shown as `Failure [INSTALL_FAILED_DEXOPT]`, while the UI only shows a generic message and a numeric error code (e.g., see Figure 2). In either case, the attack blocks the package name declared in the manifest file.

4. EXPLOITING THE ATTACK

In this Section, we first describe a proof of concept application (Section 4.1) that demonstrates our attack. Next, we show how the installation of the broken APK files can be triggered without user interaction (Section 4.2). Adversaries could use similar techniques to gain personal or financial advantage, to harm other competing developers or, to foster the installation of additional malware. In general, we envision four different scenarios: Firstly, a targeted attack against a specific competing developer's application (Section 4.3). Secondly, a multi-stage attack combining our denial-of-app attack with a so-called *master key* attack to distribute malware and disable protections on the phone (Section 4.4). Thirdly, a mass attack to block large amounts of applications at once (Section 4.5), and lastly an attack on Google's Play Store infrastructure to block an application not only from phones on which the exploit is executed, but from the market overall (Section 4.6).

[3]`adb install <filename>` returns
`Failure [INSTALL_FAILED_UID_CHANGED]`

4.1 Proof of Concept App

As a proof of concept, we developed an Android application *Denial-of-App-PoC* with zero permissions. It includes a manipulated *dex* file containing a double *Overwrite* annotation as described in Section 3.2. The *Denial-of-App-PoC* application allows the user to enter an arbitrary package name of an application to be blocked from installation. *Denial-of-App-PoC* takes this package name and the pre-built *dex* file to construct a new APK file where it places the user-specified package name in the `AndroidManifest.xml`. The `AndroidManifest.xml` is manipulated with the axml framework [16], added into the apk zip file, signed using a fixed key, and zip-aligned, so that it can readily be installed on the phone. However, as the app is broken by construction, the installation will fail and leave the secured private data folder behind. Figure 1 shows an example, where we entered the package name of the well-known Android Facebook Messenger app which we would like to block for installation.

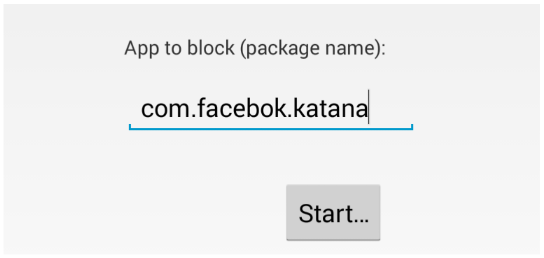

App to block (package name):
com.facebok.katana

Figure 1: Entering package name of the app which has to be blocked

After clicking on "Start..." button, the *Denial-of-App-PoC* application creates a broken app with the package name of the Facebook Messenger app. This impersonates the Facebook Messenger app, since the package name is treated as a unique id (see Section 2). *Denial-of-App-PoC* initializes the installation and the broken app gets installed, but fails with an "App not installed" error. This is everything one needs for blocking specific applications. When a user wants to install the original app now, she will fail with an error as shown in Figure 2. Since Facebook is not in the list of installed applications maintained by the Android operating system, the user cannot even try to re-install it, but remains stuck with the error whenever she tries to install Facebook.

The full source code of our proof-of-concept exploit, including the broken dex file is available at:

`https://github.com/secure-software-engineering/denial-of-app-attack`

4.2 Exploit Enhancement

With the proof-of-concept application described in the previous section, the user still needs to approve the installation of the created broken application. If she aborts the system confirmation, the attack fails. Even if she allows the installation, she afterwards gets a pop-up showing an "App not installed" error. These limitations can however be circumvented using various techniques for silent application installation as we show in this section.

Attack using infected PC.

If the attacker is for instance able to infect a computer connected to the phone, she can launch the attack from there and use the `adb install` command to silently install a broken apk on the phone. This happens without any user interaction if the Android debug mode is enabled on the device. While debugging is disabled by default, Android users who are developers on their own are likely to have it enabled, so there is still a large target audience for the attack. Examples of such multi-stage attacks have already been found in the wild, such as the Zeus malware [6].

Confused deputy installs.

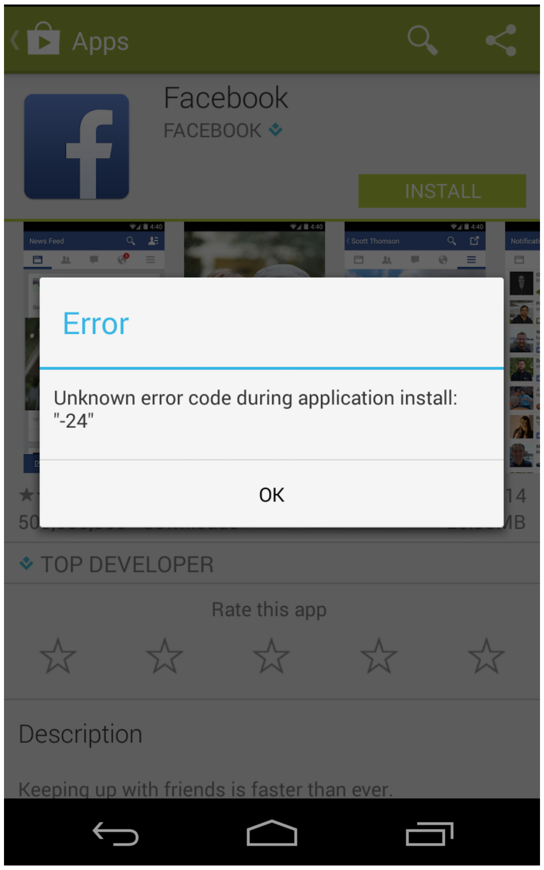
In the past years, there have also been various attacks for silently installing applications. If the Android version running on the user's phone is vulnerable to such an attack, one could also exploit it to block a massive amount of applications.

One practical way for a hidden installation and combined mass blacklisting could be a confused deputy attack [4, 8]. In general, android applications require the `INSTALL_PACKAGE` permission to silently install applications without having root privileges. This permission is however not available for third party applications; only applications signed with a system or manufacturer key can be granted this privilege. Yet, if attackers are able to hijack such a privileged application which defines the `INSTALL_PACKAGE` permission, they can abuse its permissions for installing other apps. A practical example with a proof of concept exploit was already shown on the Samsung Galaxy S3 device [14]. The attack exploits the restore function of the *Samsung's Kies.apk* application. Kies is the Samsung pendant of Apple's iTunes; it offers synchronization, backup/restore, and administration features for media files, contacts, and system upgrades. Every apk file stored in the sdcard folder `/sdcard/restore/` was installed by the vulnerable Kies' service when a restore was triggered with the sole condition that the application was not already present in the `/data/app/` folder. For triggering a restore, one only needed to send an intent with the action `com.intent.action.KIES_START_RESTORE` to this service. With our attack, such vulnerabilities can be exploited to block applications from affected phones on a large scale. The attacker simply needs to prepare a set of broken apk files and place them in the `restore` folder before triggering the recovery service. The preparation of such broken apk can be done on the fly (see 4.5). Similar techniques for hijacking applications with install privileges were also shown on various other devices from different manufacturers, for instance for older HTC phones [18].

Installs using master-key vulnerability.

Even if a confused deputy attack on an already-installed system application bearing the `INSTALL_PACKAGE` permission is not possible, attackers can still exploit the so-called master key vulnerability [9, 10]. This vulnerability allows modified system applications to pass signature verification and be installed on the phone despite the changes, even keeping the original manufacturer signature. Therefore, the modified versions can still receive highly sensitive system permissions such as `INSTALL_PACKAGE`. Even worse, there is no need for the original system application to have this permission; the modification can introduce it together with the patched code. The Android operating system will grant the new permission since the modified application is still regarded as manufacturer-signed due to the vulnerability. The App Manager contained in the Android operating system will however not show such injected permissions, leaving the user in a false sense of security and making it harder for the user to detect the attack.

We have verified that our proof-of-concept app can use this vulnerability to block arbitrary apps on Android 4.0, 4.1 and 4.2.2. We tested this on a Google Nexus S phone as well as devices from Samsung, and HTC. As a master-key vulnerability has been reported for Android 4.3 and 4.4 [10] as well, our combined attack is also applicable to these versions.


Error

Unknown error code during application install: "-24"

OK

Figure 2: Attempt to install a blocked app

4.3 Targeted Attack

The *denial-of-app* vulnerability allows arbitrary applications to be blocked, which can be exploited for a targeted attack. Imagine two competing developers offering similar games on the market and gaining revenue through advertisements. It is likely that users who cannot install one of the games will switch to the other one if both have about the same popularity or rating on the market which is quite common for many games. One developer thus can distribute a fake app, with the package name of his competitor's game. Users will just see the fake app's installation fail, and will not notice that an attack has happened. They are now, however, unable to install the competitor's actual game in the future, giving the attacker financial benefits from those users that switch to his game. The same attack vector can also be used to inhibit the installation of general popular apps in favor of malware-infected variants of those apps, fostering the installation of additional malware on the user's device.

4.4 Combined multi-stage attack

Attackers can also use the ability to block arbitrary sets of applications to complicate the detection, blocking, and removal of malware. After the user has been tricked into installing the exploit app (which can e.g., imitate or repackage a popular game), this app can abuse its system permissions gained through the master-key vulnerability to deinstall all well known anti-virus products and afterwards block them use our attack to make sure that the user cannot reinstall any of them. Afterwards, it exhibits its malicious behavior. For the user, detecting and removing such malware then requires manual investigation without any tool support. This brings the additional caveat that she will usually not suspect a replaced system app as the source of malicious behavior unless pointed to it by a scanner which is however no longer available on his phone.

4.5 Mass Attack

The exploit can easily be adapted for targeting different package names. The *dex* file containing the broken code that makes the bytecode verifier fail needs not be changed at all. One only needs to create a copy of the original exploit APK, change the package name in the manifest file, and sign it to get a customized exploit for blocking a different package name. We created a proof of concept that automates this process in an Android app which uses a blacklist of package names for directly installing them with the new exploit.

4.6 Bouncer Block

Before newly uploaded applications are allowed into the Google Play Store, they are vetted by Bouncer [15], a dynamic analysis tool. Bouncer runs the application for a defined period of time, randomly clicks on elements in the user interface and monitors the application's behavior. Only if the application neither crashes nor violates any policy checked by Bouncer, it is allowed into the Play Store. This technique was introduced to help improve the stability and security of applications available on the market.

Google's Bouncer runs on a modified Android emulator [15] and is therefore likely vulnerable to the same attacks as the emulator itself. Previous research has shown that the Bouncer's emulator is not reset after every test run [5]. We therefore suspect that uploading our exploit into the Play Store would execute the attack on this emulator, effectively blocking the target application. When the developer of the attack target then later wants to upload or update her app, and if this novel app is vetted on the same Bouncer instance, the new app will be refused until the Bouncer emulator is finally reset. In the worst case, this attack could be used to block applications from the market completely. For ethical and legal reasons we did not confirm the attack by attacking Bouncer, since this would have meant running exploits on the productive Play Store system, making the authors liable for potential damages. We did, however, confirm that our attack also works on emulators such as those used by Bouncer.

5. COUNTERMEASURES

In this section, we describe a number of countermeasures against our *denial-of-app* attack. For preventing the attack at large, the fix provided by Google as the operating system vender needs to be applied to all Android devices such that artifacts are properly cleaned after failed app installations. The original Android 4.4 version pre-installed on recent phones still contains the bug, so users are required to install the update on their own. Even worse, for phones customized by the respective manufacturers, adapted updates are often still under development or are unlikely to be provided at all as the devices have reached the end of their support period.

On smartphones running Android 4.4.2 or below, the attack cannot easily be prevented by other means than installing a fix. In managed environments such as corporate app stores, one could analyze every new application for corresponding exploit code before allowing it into the market. End-users could try to only install applications from trusted sources. Yet, this leaves the phone vulnerable to e.g., attacks abusing the debug interface from a connected infected computer.

After the system has been attacked and applications have been blocked, there are only two possibilities to remove the block. First, the user can obtain root privileges on her phone for which various tools or services exist [1, 23]. This process however often wipes all data from the phone for security reasons. After rooting the device, the user has to delete all files in the **/data/data/<package-name>** folder. "Rooting" a smartphone is however not advisable as it circumvents the Android sandbox security model and may therefore introduce new vulnerabilities which may allow applications to access sensitive data from other applications.

A second possibility is a complete factory reset of the affected smartphone which not only removes the blocking data directory, but also deletes all other installed applications and user data. This side-effect makes the solution likewise unattractive to the average user who is not in possession of a current and complete backup of all his phone data.

6. RELATED WORK

In the context of Android security, this form of attack is the first of its kind. There have however already been different attacks that exploited missing consistence checks on the apk file or insufficient of its contents such as the `classes.dex` file or the `AndroidManifest.xml` file.

Apvrille [3] has shown numerous ways to hide sensitive information in a dex file from common disassemblers such as baksmali [19] by manipulating the method index table of the `classes.dex` file. At runtime, the method is made accessible again by rebuilding the dex data structure. These attacks can however only disguise behavior and data, but cannot actively block the installation of an app.

The Android master-key exploit [9] also exploited missing apk file validations. An Android application is digitally signed and the signature is verified before installation. Normally, whenever there is a signature mismatch due to a manipulation of the file, the verification process fails and the application is rejected. In vulnerable versions of the Android OS, the signature verification and the content extraction were however implemented on different layers of the operating system. Even though both processes would unpack the apk file, the signature verification process validated different aspects than the content extraction process late extracted from the file. If the zip file table of the apk file was therefore manipulated such that there were two files with the same name, a vulnerable Android system would verify the first one but extract and install the second one. This allows an attacker to inject code files such as an additional `classes.dex` file into an application without modifying the signature. If the original application was signed using a manufacturer key, the patched application will also receive system-privileges.

One attack closely related to ours is the Denial of Service attack from Ibrahim Balic which exploited flaws in the verification process of the `AndroidManifest.xml` file [5]. He forced a system freeze by manipulating attributes in the manifest. This however just freezes the Android device and does not prevent apps from being installed as in our attack. In summary, there are many different attacks against Android, but to the best of our knowledge, the *denial-of-app* attack is a new form of attack that has not been described before.

Ratazzi et. al. [17] present a Denial-of-app attack on a multi-user Android device. If one user installs an application with a certain package name, no other user can install an app with the same package name. Only the installing user and the device owner are permitted to remove the blocking app. Our attack, on the other hand, also works for single-user devices. If applied to a multi-user device, not even the device owner can remove the block with our attack.

Ratazzi also describes preventing app installations by using upp all available app ids. This however requires installing a substantial number of apps (50,000 on a Nexus 10 with Android 4.4) which are visible in the operating system's app list and are thus highly suspicious. On the other hand, one could combine their attack and ours by installing 50,000 blocking (and hidden) apps with our approach to prevent the user from installing any new apps (not just specific ones), while at the same time making it very hard to detect and remove the cause of this global block.

7. CONCLUSIONS

In this paper, we have presented a new form of attack for mobile devices, called *denial-of-app* attack. This form of attack inhibits the installation of one or more apps on a smartphone. We have demonstrated and released a proof-of-concept exploit app which allows a user to block any application on her device. The block can only be removed in two ways, either by rooting the device and deleting specific folders in the Android file system or by resetting the device to factory defaults. Both solutions have severe drawbacks: Rooting a device forces new security problems and resetting the device deletes all installed applications and user data which is not convenient for a user (in case there is no backup). The Android security team has released a patch for this vulnerability, but to the best of our knowledge, all versions prior to Android 4.4.3 are likely to be affected by our attack.

Acknowledgements. This work was supported by the BMBF within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

REFERENCES

[1] `http://rootwiki.net/`.

[2] S. Acharya. *Samsung Confirms No Android 4.4 KitKat for Galaxy S3 and S3 Mini 3G Versions.*

[3] A. Apvrille. *Playing Hide and Seek with Dalvik Executable.* `https://www.fortiguard.com/uploads/general/Hidex_Paper.pdf`.

[4] "Automatic detection of inter-application permission leaks in Android applications". In: *IBM Journal of Research and Development (Volume:57, Issue: 6)* (2013). `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6665098`.

[5] I. Balic. *Android Vulnerability affected Google Play Bouncer (Emulator).* Blog. `http://ibrahimbalic.com/2014/android-vulnerability-affected-google-play-bouncer-emulator/`. Mar. 2014.

[6] Cert Polsks. *A PowerZeus Incident Case Study.* Tech. rep. Cert.

[7] *CVE Details.* Website. `http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224`. June 2014.

[8] A. P. Felt et al. "Permission Re-Delegation : Attacks and Defenses". In: *SEC'11 Proceedings of the 20th USENIX conference on Security.* ACM, 2011. URL: `https://www.usenix.org/legacy/event/sec11/tech/full_papers/Felt.pdf`.

[9] J. Forristal. *Android: One Root to Own Them All.*

[10] J. Freeman. `http://www.saurik.com/id/19`.

[11] E. Hamburger. *Indie smash hit Flappy Bird racks up 50K per day in ad revenue.* `http://www.theverge.com/2014/2/5/5383708/flappy-bird-revenue-50-k-per-day-dong-nguyen-interview`. Feb. 2014.

[12] International Data Corporation. *Worldwide Quarterly Mobile Phone Tracker 3Q12.* `http://www.idc.com/tracker/showproductinfo.jsp?prod\_id=37`. Nov. 2012.

[13] Lookout Mobile Security. `https://blog.lookout.com/blog/2014/04/09/heartbleed-detector/`. 2014.

[14] A. Moulu. *From 0 perm app to INSTALL_PACKAGES on Samsung Galaxy S3.* `http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html`. 2012.

[15] J Oberheide and C Miller. "Dissecting the android bouncer". In: *SummerCon2012, New York* (2012).

[16] Panxiaobo. *axml.*

[17] P. Ratazzi et al. "A Systematic Security Evaluation of Android's Multi-User Framework". In: *IEEE Mobile Security Technologies workshop (MoST), 2014.*

[18] T. Relph-Knight. `http://www.h-online.com/security/news/item/Android-holes-allow-secret-installation-of-apps-1134940.html`. 2010.

[19] smali. *smali: An assembler/disassembler for Android's dex format.* Google Code. `https://code.google.com/p/smali/`. June 2014.

[20] M. Smith. `http://www.engadget.com/2012/09/25/dirty-ussd-code-samsung-hack-wipe/`. 2012.

[21] R. Vallée-Rai et al. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research.* CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–. URL: `http://dl.acm.org/citation.cfm?id=781995.782008`.

[22] M. de Vries et al. "POPSIS - Pricing Of Public Sector Information Study". In: *European Commission Information Society and Media Directorate-General* (2011). `http://www.epsiplatform.eu/sites/default/files/apps_market.pdf`.

[23] WugFresh. `http://www.wugfresh.com/`.

[24] W. Zhou et al. "Fast, Scalable Detection of "Piggybacked" Mobile Applications". In: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy.* CODASPY '13. 2013.