Plugin in the Middle - Minimising Security Risks in Mobile Middleware Implementations

Peter Aufner, Georg Merzdovnik, Markus Huber and Edgar Weippl
SBA Research
Sommerpalais Harrach, Favoritenstrasse 16, 2. Stock, AT-1040 Vienna, Austria
{paufner,gmerzdovnik,mhuber,eweippl}@sba-research.org

ABSTRACT

Mobile computing platforms, like smartphones and tablet computers, are becoming a commodity nowadays. The diversity and fast changing nature of these systems often makes it hard for developers to adapt their applications to the user's context. To simplify development a number of approaches have been suggested, which offer a context-middleware solution such that common functionality can be pooled into plugins and provided to applications. These extensions are then automatically installed if needed, thus enabling easier and faster development of complex applications. Furthermore, if the device changes, it often suffices to exchange the plugins for the applications to function correctly. However, mobile platforms like Android never expected integration in the sense that one application would dynamically host pieces of code from different vendors and allow access to other applications, since doing so basically circumvents many built-in security measures of the operating system. In this paper we analyze Ambient Dynamix, an advanced context-middleware solution, in detail. Hereby, we propose and evaluate security mechanisms to increase the security of Ambient Dynamix. We outline a system to verify the permissions an application requests against the actual Ambient Dynamix plugins it uses. In the following, we evaluate the use of static code analysis to ensure requested and used permissions by a novel method for lightweight on-device analysis. Finally, we propose a secure infrastructure to host, download and install third-party plugins. Our proposed security extensions significantly improve the user's security regarding third-party applications and considerably advance the area of secure mobile middleware.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Security, Privacy

Keywords

mobile middleware, plugin security, Android

1. INTRODUCTION

Mobile devices play an important role in the daily life of many people nowadays. They have become their central information interchange and provide a broad foundation for the Internet of Things (IoT). However, the many uses of such devices as well as the heterogeneity of systems and environments increase the complexity for developers to adapt their applications to the growing needs and settings of their users. A way for software to adjust their behaviour to these ever changing conditions is to incorporate contextual information extracted from the user's environment. To simplify the development of context-aware applications typically a dedicated middleware is implemented which reduces the complexity of context-modeling for developers. However, since these frameworks are often difficult to use or lack certain features like runtime updates or are generally hard to extend, Ambient Dynamix, a plug-and-play context framework has been introduced by Carlson et al. [4]. In order to solve the described issue of other middleware implementations, the Ambient Dynamix project has recently started the development of a middleware solution that aims to encapsulate more common functionalities than the APIs of the Android mobile operating system provides. The project is open sourced and allows anyone to develop plugins. Developers can apply for hosting with the official repository or create their own repositories. The middleware uses a modern software architecture to utilize the resources the Android operating system offers, along with Open Services Gateway initiative (OSGi) technologies to efficiently handle the encapsulation of various plugins.

The openness and extensibility of Ambient Dynamix has a serious impact on the security of the whole mobile system. Since Ambient Dynamix itself is treated as one of many apps by the Android operating system, there are conflicts with the native security framework of the Android operating system that need to be resolved. Android will regard any plugin that is run inside Ambient Dynamix as being Ambient Dynamix. Hence, if there is a malicious plugin, Android will not prevent it from running. Without proper security measures, a single plugin could lead to complete compromise of the mobile device. This could range from a simple leakage of the device owner's contact information to rooting of the device and the installation of a backdoor. In order to address these challenges, we are extending the Ambient Dynamix framework with security features to prevent abuse of the system and to provide additional protection of the user's privacy. The main contributions of this paper are as follows:

- We evaluate an on-device method to prevent applications from misusing plugins for privilege escalation

- We evaluate a lightweight and automated analysis method to prevent execution of malicious plugins
- We propose a secure content framework for plugin and application distribution

The remainder of this paper is structured as follows. At first we provide an overview on related work and important background concepts in Section 2. In Section 3 we outline the existing Dynamix framework and propose a middleware security concept, followed by the evaluation of the described system in Section 4. We discuss our findings in Section 5 before we conclude our work in Section 6.

2. BACKGROUND AND RELATED WORK

This section covers information on the used technologies as well as related work concerning the security implications of middleware framework implementations on mobile devices.

As described previously, Ambient Dynamix was chosen as a base for this work because of its rich feature set, openness and community based approach. Ambient Dynamix is described by the authors as follows [4]: "Ambient Dynamix (Dynamix) is a lightweight software framework that enables mobile apps and websites to fluidly interact with the physical world through advanced sensing and actuation plugins that can be installed on-demand." Figure 1 outlines how Ambient Dynamix communicates with the Internet, sensors and other applications as well as the important aspects of the middleware's software architecture:

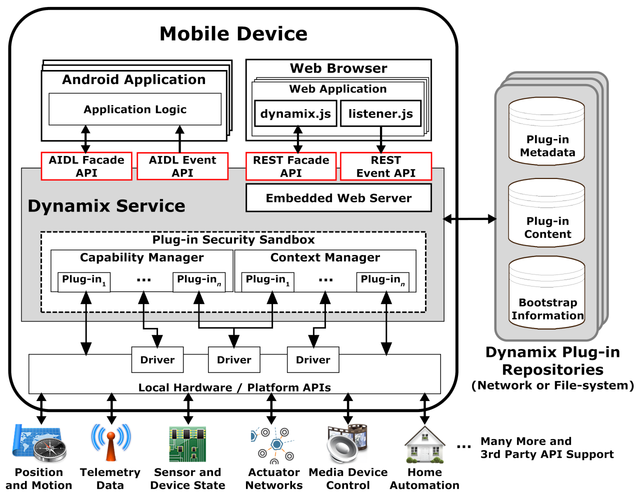

Figure 1: The architecture of Ambient Dynamix as shown in [7]

Ambient Dynamix is based on OSGi, which is a mechanism to dynamically manage code packages inside Java applications. It allows keeping functional units separated and provides mechanisms to easily load them during runtime as well as to define dependencies between them. Ambient Dynamix' OSGi implementation is based on Felix[1] by the Apache Foundation. Although OSGi can preserve the sanity of the host infrastructure, three main attack vectors to insert malicious code exist: either during development, by creating malicious bundles or by creating counterfeit repositories. Parrend et al. [14] developed a toolsuite that supports the whole OSGi signature standard as well as deployment of bundles. Their toolsuite ensures that developers can be held accountable for the software they release. The proposal of Parrend et al. has not been included in the main release of OSGi so far. Phung and Sands [15] investigated methods to improve the security of OSGi-based environments with Aspect Oriented Programming (AOP). Based on AspectJ, a well-known AOP solution, they implemented a reference monitor-style policy set and different history-based security states. They also identified actions that are not malicious by themselves, but could have malicious effects if repeated often. One example for such an action could be an application that utilizes a cellular gateway to send a vast number of SMS messages. This enables their framework to suppress, replace, insert and truncate security-relevant actions. Ambient Dynamix relies on standard Android APIs for Inter-Process communication. Plugins can be called from other applications by so called Intents, which provide means for combining code in different applications during runtime. By the nature of encapsulating the whole functionality of different plugins, Dynamix is forced to break the permission system of Android in order to allow the plugins to work. In order to use the framework, an application is required to implement the Ambient Dynamix API and request the use of a specific plugin from the Ambient Dynamix plugin repository. If the plugin is already installed on the system, permissions will be asked once and afterwards the plugin will be executed on behalf of the calling application. Otherwise the required plugin can be obtained from either the official, a third-party or a local repository hosted on the SD card. The overall design of Ambien Dynamix leads to a number of security issues. Sbırlea et al. [17] aim to automatically detect three different types of attacks. *Permission Collusion* is the case where an application raises its privileges by using another, more privileged application. In the *Confused Deputy Attack*, an unauthorized caller invokes protected actions in another application. And last the *Private Activity invocation*, where activities that were not meant to be called from the outside are activated from another application. These three attacks describe the most common kinds of attacks against the Intent system. They also developed a system to check applications for these weaknesses and during their analysis discovered that several applications could be abused to leak sensitive information such as location data. A similar project has also been completed and described by Mann and Starosim [12]. They described a framework for static detection of privacy leaks in Android applications. The current architecture of Ambient Dynamix does not address cases where applications try to use plugins without having the necessary permissions. Currently, the framework only verifies permissions based on the requirements reported by the application's developer and no further checks are employed. In addition, the current design of Ambient Dynamix has a number of important implications on the usefulness of the Android permission system itself. The main problem is, that applications can use the framework without requesting the correct permissions from the operating system. Bridges et al. [3] have examined the gap between security privileges requested by Android applications and the ones they actually use. According to their findings, popular apps like Twitter are able to run underprivileged. This happens despite the fact that the developer documentation claims that any API call must have the proper privileges reflected in their manifest. However, they put this finding in relation as it could also be caused by carelessness of developers. In fact, the app crashes during runtime if the non-privileged action is executed. If a developer adds some functionality that requires the permission but never uses it, then no errors would arise during execution. Another problem that falls into the same category are over-privileged applications, which are tackled by Porter-Felt et al. [10]. The findings of Porter-Felt et al. are important for the understanding of the Android se-

[1]`http://felix.apache.org/`

curity system, as the case of over-privileged apps illustrates how the permission checking on Android actually works. Their findings imply that it is nearly impossible to re-implement a permission check that is feasible in user space. Vidas et al. [18] further examined this problem and developed an Eclipse plugin that checks which permissions are necessary for an application to run before it is deployed. They created their database by parsing the Android developer documentation, unfortunately, they did not go into any details on how their final permission map was obtained. Ambient Dynamix allows code reuse in a way that would not be possible on Android, unless major changes to the whole application concept of Android are made. For example, Ambient Dynamix provides APIs that cover many important functions to access various services and devices inside the system. One drawback of the current Ambient Dynamix framework is the lack of a central package manager for plugins and libraries. The concept of a central package manager is successfully used in most Linux distributions. A centralized package manager would increase the administrative burden compared to the current model in Android with self contained applications, but it would allow for easier methods to reuse code and libraries. This could lead to faster development of complex context aware applications as well as easier deployment and maintainability on different devices.

3. SECURE MIDDLEWARE FRAMEWORK

This section describes the security improvements we designed and evaluated for Ambient Dynamix. Firstly, we describe how permissions of applications are verified in order to prevent misuse of the system. Secondly, we propose a method to analyse plugins in order to prevent privilege escalation. Finally, we describe a secure architecture for extension repositories in order to prevent tampering by verifying the integrity of downloads.

Ambient Dynamix introduces a concept to the Android system that is rather different from other applications. With the plugin based infrastructure in mind it basically needs to reimplement the Android permission system without access to root privileges, and therefore without modifications to the mobile operating system itself. To work around this limitation, Dynamix requests the combined set of permissions from the operating system that are required by plugins. Since this yields a lot of power to Dynamix, the system has to ensure that plugins and applications only use the permissions that they have requested in order to prevent abuse by extensions.

3.1 Enforcing Permissions on Calling Apps

Prior to this work Ambient Dynamix included only a rudimentary permission system. The sole purpose was to inform the user of the permissions that an application wanted to use. Since these permissions were stated by the developer and were not checked, they had no influence on the actual execution of the software. This means that a developer was able to use every permission that has been granted to the Dynamix Framework without the user ever noticing it. Since Dynamix is heavily community centered and allows the setup of third party repositories, another problem is that server side checks would not be sufficient to protect the user. Therefore we decided to develop a client side solution to prevent misuse of the system. The main idea is to prevent applications from using plugins which provide more permissions than the application originally requested from the Android system. Checks are thus required which prevent privilege escalation by applications. This means that a Dynamix plugin should only be available to an application, if the application has at least requested the same system permissions as the plugin.

Implementation: To check an application for used permissions, we make use of the Android permission system. Ambient Dynamix is called by an Intent from another application. The action creates a Java object, which amongst other things, contains the unique id (UID) of the calling application. With this key it is possible to uniquely identify an application. By acquiring the Android PackageManager from the global context, it is possible to get all the important information of any application that is installed on the system. This allows us to gather a list of all the permissions an application has requested. The implementation for gathering application permissions is outlined in Listing 1. With the list of collected permissions the Ambient Dynamix framework can decide whether the application has acquired sufficient privileges from the system in order to call the required plugin.

```
// Construct a new application for the
   caller
ApplicationInfo info = null;
// get the PackageManager
PackageManager pm = context.
     getPackageManager();
// List packages by given id, should
     only be one
String[] packages = pm.
     getPackagesForUid(id);
PackageInfo pkgInfo = null;
try {
// get the ApplicationInfo for the
     chosen application
  info = pm.getApplicationInfo(
       packages[0], PackageManager.
       GET_UNINSTALLED_PACKAGES);
// Finally, grab only the PackageInfo
     we are really interested in
// We need to include the
     GET_PERMISSIONS flag to introspect
       app permissions
  pkgInfo = pm.getPackageInfo(packages
       [0], PackageManager.
       GET_PERMISSIONS);
} catch (NameNotFoundException e) {
  Log.e(TAG, ''Could not get
       information for calling UID: ''
       + e.getMessage());
}
return null;
```

Listing 1: Programmatically getting app permissions

3.2 Validate Permission Use

In the previous section we introduced methods to ensure that calling applications are not able to escalate their privileges through Ambient Dynamix. In this section of the paper we focus on validating that plugins can only perform actions for which they have permissions. Because of the inherent openness of the system, we are facing the same problems as before with public third party repositories. Therefore the plugin's use of permissions needs to be validated. Furthermore the solution needs to have a reasonable low runtime, in order not to delay installation of new plugins. Since the Dynamix framework enables us to prevent the use of reflection and other dynamic code loading techniques in plugins it suffices to develop a static analysis approach. The analysis works by iterating over the used library calls inside the plugin and by checking which permissions are needed to carry out the operation. If a plugin uses library calls which it had not requested permissions for, the installation should abort and the user should be notified. In order to implement these checks, the permissions required by certain library calls have to be identified first. The Android developer documentation [11] provides all the information required to map certain classes and methods to corresponding permissions.

Implementation: To generate a map of classes and methods with corresponding permissions a custom crawler using Scrapy[2], a Python web-crawling framework was implemented. Certain methods may be called with different types of parameters but require the same permissions, or a class might occur twice in the list. To filter out these double results an additional pipeline was employed. It remembers all the processed items and compares them to each new one. If no collision is detected, the object will be added to the final result, otherwise it will be discarded. Scrapy also supports output in the JavaScript Object Notation[3] format, which can easily be processed in Java. This solution allows for a quick and easy way to get a set of all permissions needed by classes and methods. Once set up, it can be run whenever a new version of Android is released to automatically correct the permission map. The permission map generated from the Android developer documentation contains a total of 308 records. These can be divided into a set of 277 unique Methods and a total of 31 classes that require a certain set of permissions to be used. After gathering the permission map, we can use this information for the client-side static analysis. The fact that code for Android is compiled to a different bytecode than plain Java code renders most supporting libraries and utilities useless. This holds especially for techniques like runtime weaving, where the bytecode is dynamically modified. Due to the completely different virtual machine, Dalvik, on the Android operating system, standard Java libraries for bytecode modification can not be used directly and need to be rewritten. Fortunately, an Android version of the asm library, called Asmdex [1], has already been developed. It allows iterating over all classes that are defined inside an application. Asmdex implements the visitor and callback patterns. It iterates through all the classes in the given dex file and calls the respective method of the subscriber whenever a certain criteria is met. In this case we are interested in all classes and methods being visited in order to check their corresponding permissions. A global list keeps track of each method and class uniquely. We can use this information to compare the permissions a plugin requests, to what it actually requires based on the permission map. Furthermore, any given set of methods or classes can be prevented from being used. This can be extended to block the use of unwanted methods or functionality. This method is also robust against obfuscation, since at some point, the API calls will have to be disclosed.

3.3 Authenticating Repositories and Packages

Ambient Dynamix is an open, community based system, where plugins and applications can be installed from a central as well as third-party repositories. The problem with the current state of the framework is that there is no way to validate downloaded packages. Since currently not even SSL is in use on the download servers, it would be easy to intercept and exchange packages with a standard man-in-the-middle attack. Therefore it is important to properly authenticate the used repositories. This way, we prevent spoofing attacks and also provide a way for integrity checks of the packages. To implement these checks we propose to cryptographically sign the repository index files as well as each package individually.

Implementation: Java and Android provide ways to sign .jar and .apk files respectively. For Android applications it is even a requirement for .apk files to be digitally signed in order to execute them. However, these mechanisms cannot be leveraged in the case of Ambient Dynamix. The .apk file signature is only checked, if an application is installed and executed by Android. Since the plugins are dynamically executed from within Ambient Dynamix, this protection mechanism is bypassed. The final solution is based on BouncyCastle [8], a Java cryptographic library. This makes it easy to work with PGP keys, and allows for signing and verifying of file signatures on the client. For each repository a public/private key pair is created. The public key is distributed to the clients to allow them to verify the signatures generated with the repository's private key. The server signs its index file, and the client only adds the repository, if the signature can be verified correctly. Every time a user downloads a plugin, the system also verifies if it was signed with the correct key, and indeed was distributed by the given repository. Furthermore, this signature can also be used to employ integrity checks to prevent corrupted files from being installed.

4. EVALUATION

This section describes the performance evaluation conducted to test the implemented methods in a realistic setting.

4.1 Test Setup

To check the correctness and the performance of the solutions, four Android devices of different kinds were used. The devices are listed in Table 1. Each phone is based on a different hardware platform with different specifications. Two devices used a test build of CyanogenMod 10.2[4], a customised Android distribution. The other two mobile phones were running on Android 4.3 Jelly Bean[5].

4.2 Results

4.2.1 Application Permission Use

The implementation of the application permission use verification is given in Listing 1. Since these checks do not bear any algorithmic difficulty when being executed, the performance of the implementation was not evaluated separately. To assert the correct functionality, the source of the Dynamix Logger sample app, available at [4], was modified. Additionally the barcode scanner plugin and the Wi-Fi scanner plugin were downloaded from the official repository and added to a local repository. Using a local repository allowed us to declare arbitrary permissions for the applications in use. By tampering with the requested permissions we were able to evaluate correct functionality of our implementation. Without changes the barcode scanner requires access to the camera, while the Wi-Fi scanner obviously needs access to the network configuration. The following scenarios were used to validate the correctness of our implementation:

- **Test cases for a valid execution:** The calling application has at least all necessary permissions.
- **Test cases for blocked execution:** An additional permission was unnecessarily declared for the barcode scanner but the calling application had not requested this permission (over-privileged plugin).
  Additionally, the calling application might also lack the necessary permissions for a specific plugin.

Our extended framework passed all the tests successfully.

[2]`http://scrapy.org/`
[3]`http://www.json.org/`
[4]`http://www.cyanogenmod.org/`
[5]`http://www.android.com/about/jelly-bean/`

Table 1: Device Specifications

Device	Processor	Clock Speed (GHz)	RAM (GB)	OS Version
Google Nexus 4	Qualcomm Snapdragon S4 Pro APQ8064	1.5	2	4.3 Build JSS15J
Asus TF700T	Nvidia Tegra 3 T33	1.6	1	CM 10.2
Samsung Galaxy S3	Exynos 4412	1.4	1	CM 10.2
Google Nexus 7	Nvidia Tegra 3 T30L	1.2	1	4.3 Build JWR66Y

4.2.2 Plugin Permission Checks

In order to assert the correctness of the solution, it was tested with the barcode scanner plugin and the zephyrhxm plugin from the official Ambient Dynamix repository [5]. Hereby, we verified if all permissions are detected by our security extension and if these detected permissions match the plugin declarations. We could successfully verify the permission set of both plugins. In order to assess the performance of our solution, four test runs were conducted on each device. For each run, the time to load the permission table as well as the time needed to analyze the two plugins was taken. The table below shows the averages of the four test runs for each device:

Table 2: Average durations of four test runs for permission checks in milliseconds (ms).

Device	Permissions (ms)	zephyrhxm (ms)	barcode (ms)
Nexus 4	159.25	1,912.25	25,836.5
TF700T	168.5	1,434.75	19,825.25
i9300	65.75	1,091.25	13,720.25
Nexus 7	143.25	1,717	21,745.5

The results in Table 2 show that while the solution seems to be rather slow to be executed on every single plugin, it nonetheless serves the purpose of securing the environment from untrusted plugins. It should also be noted that the barcode plugin is one of the larger plugins found in the official repository at [6], where the median of the plugin sizes is at 39kB. So it should be expected to have results closer to the run-time of the zephyrhxm plugin rather than the barcode plugin. Furthermore the checks are only performed when a plugin is newly installed or updated. This means that only the installation time is affected by the checks, which is a small trade-off compared to the gained security.

4.2.3 Repository Validation Implementation

The main complexity of the provided implementation for this library lies within the use of the available cryptographic library, Bouncy Castle. The final implementation was inspired by the Apache Commons OpenGPG project, but was further improved due to the lack of certain functionality. To work with the keys, a structure to store PGP keyrings is necessary. The public and private keyring files need to be imported. Once this is done, the signature verification process can begin. Based on the Apache code, a streaming signature verifier was wrapped to be used with a regular file input. The complexity of the implementation lies within the streaming signature verifier. This class is instantiated with the signature of the key to check against, as well as the keyring that was created before. After initialisation it is provided with the bytes of the file in order to perform the signature check, which is utilizing the BouncyCastle cryptographic library. In order to assess the performance of the solution, four test runs have been conducted on each device. For each test run the time to prepare the GPG public keyring for use was measured. Table 3 shows the averages of the four test runs for each device. Since the keyring only needs to be loaded once before any checks occur, this solution is clearly feasible to be run during the usual installation process of a plugin.

5. DISCUSSION

5.1 Enforcing Permissions on Calling Apps

The whole concept of Ambient Dynamix enables the creation of applications that do not require any permissions. By using the provided plugins, which already received the required permissions from the Android system, it would be possible to run a huge set of operations without informing the user about the application's capabilities. This concept of operation has been introduced by Russello in [16]. However, introspecting the set of permissions of an application, which is calling a method via an intent has not been discussed so far. The Android operating system provides methods which allow us to achieve this functionality in a stable and robust way, since it relies solely on core features of the operating system. Nonetheless, it might be a good idea to add a dedicated permission to the Android system to inform a user about the use of Ambient Dynamix in downloaded applications. The assertion that calling applications have a matching set of permissions with the plugins they use should only concern the developer. Therefore it is important to employ an ecosystem which allows notification of application developers about permission changes prior to plugin update releases. This is required to give them a grace period during which they can update their application in order to request the required permissions from the operating system. If a developer fails to release a new build of the application that adheres to the standards and requires the correct permissions, it must be ensured that the user is informed of the developer's fault. A suggestion needs to be made, to contact the developer and to point out the problem.

5.2 Asserting Permissions of Plugins

The main motivation for this part of the work was to prevent plugins from executing unexpected behaviour. Bartel et al. [2] proposed a solution to this problem by modifying the binary on the device. Their results have shown that this is a very slow process, especially for large sized binaries. Relying on binary modification to enforce desired behavior of an application can also bear other problems. On the one hand this could corrupt the logic of the application as well as fail to prevent undesired behavior. Certain techniques like reflection could probably still be used to prevent detection of malicious methods. This means that it is necessary to reduce the developers capabilities in using such techniques in their applications. The preferred solution to solve this task would have been to use debugging techniques directly on the device. The problem is, that Android mainly relies on commodity hardware for development purposes and only provides a limited set of debugging possibilities on the

Table 3: Average durations of 4 test runs for GPG signature checks in milliseconds (ms).

Device	read keyring (ms)	barcode correct (ms)	zephyr correct (ms)	barcode bad-GPG (ms)	barcode bad-jar (ms)	zeyphr with barcode GPG (ms)
Nexus 4	107.75	112.75	59.25	56	82.5	61.25
TF700T	57	137	62.75	64	82.75	75
i9300	67	67.25	47	43	66.75	44.5
Nexus 7	108.5	115.5	85.25	69.75	88	72.5

device. Complex solutions that require rooting or enabling specific settings are also out of the scope for a novice user. Another solution would have been to employ an emulator. This would allow extensive behavioral analysis on the various applications. This way it would also have been possible to allow and test applications that use the native development kit to run C code or make extensive use of reflection. The problem here is that the user would need to trust third party repositories to extensively analyse the provided plugins. Finally, a robust solution utilizing static code analysis was developed. Since certain coding techniques are blacklisted in the Dynamix Framework, this suffices to recreate the permission checks Android would perform, if the code was run as a normal application.

5.3 Authenticating Repositories

When looking at ways to authenticate repositories, open source communities like OpenSuSE [13] and Debian [9], have already provided good implementations. These solutions are based on GPG public key cryptography to authenticate repositories to users. Usually it is employed to sign repository index files. Furthermore checksums are provided to validate the integrity of downloaded packages, although it is open for discussion if these additional checksums are really necessary. For our implementation the lightweight approach without checksums was chosen to keep the system simple. This also suffices to detect faulty transmissions, but mainly serves the purpose of application authentication. Repository authentication is an important feature to ensure that downloaded applications and plugins cannot be replaced by a malicious third party. When we take a look at the browser world, though, it becomes obvious that it is not that easy to make users aware of the purpose of this security feature. A good basis for the chain of trust is to ship Ambient Dynamix with a set of keys for the official repository. The users automatically receive genuine software, as long as they use the official repository. When they decide to use third party repositories they need to manually verify the corresponding keys. This use case is similar to existing systems like the described Linux distribution software repositories.

6. CONCLUSION

In this paper we present our work on providing a secure and usable framework for context-aware Internet of Things (IoT) applications. Our work consists of three major contributions to considerably improve the security of the existing Ambient Dynamix framework. The first non-trivial problem we tackled consisted in a method to prevent privilege escalation of applications by introspecting the calling applications on requested permissions. This was achieved by leveraging the Android permission system. Our evaluation shows that the proposed method is feasible in practice and cannot be circumvented without tampering with the mobile operating system itself. Next we showed that, with a reasonable tradeoff between functionality and performance, it is feasible to implement on-device checks of downloaded plugins to prevent misuse of the middleware. By checking all the classes and methods that are used inside a plugin, it is possible to robustly determine if a plugin is requesting exactly the permissions it actually requires. It also allows to block plugins that use unwanted methods, like writing arbitrary files to the storage, or using reflection to load additional code that was not visible to the static code analysis. The performance tests showed that the method is feasible given the expected average size of a plugin to be used with Ambient Dynamix. Furthermore the checks only need to be run during the installation of the plugin. Taking this into account the waiting time seems to be acceptable compared to the increased security. Finally we presented a usable architecture for authenticating packets and repositories on a custom Android middleware system. We proposed and implemented a portable solution based on the well established Bouncy Castle cryptographic library. Our solution builds upon public key cryptography. A basic set of keys can be distributed along with Ambient Dynamix. This basic set of keys would provide a basis for the chain of trust and could also be extended by additional keys for community repositories. Our evaluation focused on one specific mobile middleware framework and we believe that our findings from the basis to secure state-of-the-art mobile middleware.

Acknowledgements

This research was funded by COMET K1, FFG - Austrian Research Promotion Agency. Moreover this work has been carried out within the scope of u'smile, the Josef Ressel Center for User-Friendly Secure Mobile Environments. We gratefully acknowledge funding and support by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG and NXP Semiconductors Austria GmbH.

7. REFERENCES

[1] Asmdex: `http://asm.ow2.org/asmdex-index.html`, last accessed 09/09/2013

[2] Bartel, A., Klein, J., Monperrus, M., Allix, K., Traon, Y.L.: Improving privacy on android smartphones through in-vivo bytecode instrumentation. CoRR abs/1208.4536 (2012)

[3] Bridges, E., Kishore, D., Pedo, J.: Android app security analysis. Governor's School Of Engineering And Technology Research Journal (2012)

[4] Carlson, D.: Ambient dynamix homepage, `http://ambientDynamix.org/`, last accessed 09/09/2013

[5] Carlson, D.: Ambient dynamix main repository, `http://repo.ambientDynamix.org/Dynamix/context_plugins/live8/`, last accessed 09/09/2013

[6] Carlson, D.: Ambient dynamix main repository version 9, `http://repo.ambientDynamix.org/Dynamix/context_plugins/live9/`, last accessed 09/09/2013

[7] Carlson, D.: The architecture of ambient dynamix,

http://ambientDynamix.org/documentation/Dynamix-overview, last accessed 09/09/2013

[8] Castle, B.: http://www.bouncycastle.org/java.html, last accessed 09/09/2013

[9] DebianWiki: Setting up signed apt repository with reprepro, http://wiki.debian.org/SettingUpSignedAptRepositoryWithReprepro, last accessed 09/09/2013

[10] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. CCS '11, ACM (2011)

[11] Google: Android developer documentation, https://developer.android.com/guide/components/index.html, last accessed 09/09/2013

[12] Mann, C., Starostin, A.: A framework for static detection of privacy leaks in android applications. In: ACM Symposium on Applied Computing. ACM (2012)

[13] OpenSuSEWiki: Secure installation sources, http://old-en.opensuse.org/Secure_Installation_Sources, last accessed 09/09/2013

[14] Parrend, P., Frenot, S.: Supporting the secure deployment of osgi bundles. In: WoWMoM. pp. 1–6. IEEE (2007)

[15] Phung, P.H., Sands, D.: Security policy enforcement in the osgi framework using aspect-oriented programming. In: COMPSAC'08. pp. 1076–1082. IEEE (2008)

[16] Russello, G.: Android security, http://www.cs.auckland.ac.nz/compsci725s2c/lectures/Russello-Smartphone-Security.pdf, last accessed 09/09/2013

[17] Sbırlea, D., Burke, M.G., Guarnieri, S., Pistoia, M., Sarkar, V.: Automatic detection of inter-application permission leaks in android applications. Technical Report TR13-02

[18] Vidas, T., Christin, N., Cranor, L.: Curbing android permission creep. In: W2SP'11 (May 2011)