POSTER: The ART of App Compartmentalization

Michael Backes
CISPA, Saarland University
& MPI-SWS
Saarland Informatics Campus

Sven Bugiel
CISPA, Saarland University
Saarland Informatics Campus

Jie Huang
CISPA, Saarland University
Saarland Informatics Campus

Oliver Schranz
CISPA, Saarland University
Saarland Informatics Campus

ABSTRACT

On Android, advertising libraries are commonly integrated with their host apps. Since the host and advertising components share the application's sandbox, advertisement code inherits all permissions and can access host resources with no further approval needed. Motivated by the privacy risks of advertisement libraries as already shown in the literature, this poster introduces an Android Runtime (ART) based app compartmentalization mechanism to achieve separation between trusted app code and untrusted library code without system modification and application rewriting. With our approach, advertising libraries will be isolated from the host app and the original app will be partitioned into two sub-apps that run independently, with the host app's resources and permissions being protected by Android's app sandboxing mechanism. ARTist [1], a compiler-based Android app instrumentation framework, is utilized here to recreate the communication channels between host and advertisement library. The result is a robust toolchain on device which provides a clean separation of developer-written app code and third-party advertisement code, allowing for finer-grained access control policies and information flow control without OS customization and application rebuilding.

1. MOTIVATION

To support free distribution, advertising libraries are often included in Android apps. An advertising library is a pre-compiled third-party package and can be easily integrated with the host app by a set of documented interfaces. Since those libraries are not developed by the app developer, they are considered as untrusted code and have the potential of privacy breach, as shown in [2]. Android employs an application-based coarse-grained permission model to control the usage of sensitive APIs. Thus all app components, including those from third-party libraries, will share all the requested permissions. Consequently, permission protected data like the current locations, contact list and device information can be easily gathered and exfiltrated to the advertising network. Considering the importance of allowing free-to-download apps in Android ecosystem, we do not intend to block advertisements. One of the measures suggested to mitigate this unsatisfactory situation is app compartmentalization, where an app is split into multiple security principals and each principal has its own set of privileges.

State of the Art. Existing app compartmentalization approaches can be classified into two categories: In-App privilege isolation and Inter-App privilege isolation. In-App privilege isolation mechanisms [3, 4] establish trustworthy boundaries between different components within an app. Software Fault Isolation (SFI) and Hardware Fault Isolation (HFI) are the common techniques which are applied to achieve memory separation. Both of those techniques involve complicated memory and register operations, which may result in compatibility issues due to faults. Inter-App privilege isolation mechanisms [5, 6, 7] make use of the Android app sandbox to draw a system-level trust boundary between the host app and the advertising library. The advertising library will be moved from the host app and run as a standalone app or service. The process-isolated sandbox on Android can effectively protect the corresponding permission sets and app resources from being violated. However, recent implementations of these approaches require system modification and application rewriting. System modification renders it inapplicable for regular non-tech savvy end users. And application rewriting results in app signature missing, which breaks the signature-based same origin model of app and makes the app fail to auto update any longer.

Our Ongoing Work. In contrast, we aim for a system-level privilege isolation mechanism that enhances the privacy management of advertising supported apps while not requiring any system modification and retaining app updates. We leverage the ART-based instrumentation framework ARTist [1], which supports code injection in the process of conducting ahead-of-time compilation on device, to strip the library from the app and reconnect all communication channels using an IPC mechanism that reintegrates the library code across app boundaries. By extracting the library code into a separate app, we effectively transform the problem of intra-app privilege separation to inter-app privilege separation, which is guaranteed by Android's app sandboxing. This compiler-based solution is realized as a standalone app. It can compartmentalize the trusted host components and untrusted advertising components to lay

CCS'16 October 24-28, 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4139-4/16/10.
DOI: http://dx.doi.org/10.1145/2976749.2989056

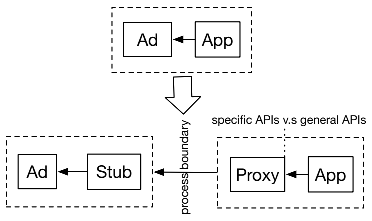

Figure 1: Inter-App Communication Mechanism.

the foundation for fine-grained permission management and app resources protection without OS modification and application rewriting.

2. DESIGN DECISIONS

2.1 Target Third-Party Libraries

While in theory our approach is applicable to a wide range of third-party libraries, we only focus on advertising libraries for the following four reasons. First, as already indicated above, third-party advertisement libraries are known to pose privacy violation risks to users. Second, advertising libraries build a large fraction of all libraries included in Android apps. App developers mostly include advertising libraries to support free installation. The results from our ongoing library detection project show that 25.94% of 100,712 application samples embed Google Play Services Ads. Third, the fragmentation of advertising libraries is rather small. According to the same study, the installation of Google Play Services Ads ranks first in the advertising library list, followed by Flurry with 17.85% installations. After that, the percentage decreases dramatically to 3.11% for Amazon Ads which ranks tenth. So we can support an overwhelming majority of apps by just targeting a small number of libraries with limited efforts. Fourth, according to the documented APIs of top 10 libraries and real app analysis, the interaction model between the host app and the advertising libraries is well-defined with a low interaction frequency. Low interaction frequency means acceptable instrumentation efforts on compartmentalization and controllable communication expense at runtime. In contrast, libraries like Guava are deeply integrated into host apps, which makes enforcing compartmentalization hard.

2.2 Inter-App Communication

In the current model, advertisements are imported into their host apps by importing the compiled packages. The host app and advertising libraries are intertwined through well-documented APIs of libraries, e.g., sync life-cycle or load and display advertisement. In this work, those advertising libraries will be moved to a standalone process entirely. In order to recreate the broken communication channels between host app and library code, Android's IPC mechanism is utilized to bridge the gap between the now distinct apps. The result is that the app is still able to display advertisements as part of its views, while keeping the advertisement code behind a strong security boundary.

The communication interfaces are defined by Android's Interface Definition Language (AIDL) beforehand, together with a pair of communicating proxy and stub implementations. As describe in Figure 1, the host requests the advertising service through methods defined in proxy, triggering a remote procedure call. The proxy redirects this request to the corresponding stub through AIDL interfaces and returns the result. Concerning the AIDL interface definition, there are two optional choices:

Library-Specific AIDL Interfaces. Generating a set of AIDL interfaces for each advertising libraries is a straightforward solution. The interfaces are constructed according to the documented library APIs in a one-to-one mapping. From the host app's view, it can just invoke the proxy method like the original library. The increased effort of AIDL generation for each library is traded for minimal instruction modification effort.

General AIDL Interfaces. Generating a set of general AIDL interfaces valid for all advertising libraries is a more generic solution. Only a set of general information transmission interfaces is provided in this approach. The host bundles all the advertising service invocation information and sends it to the advertising app. At the advertising service side, the received information is parsed and the requested operation is executed using Java reflection APIs. While this approach requires more instruction analysis and modification work, it does not request explicit per-library models.

In our ongoing work, both solutions will be implemented and evaluated to find the one best suited for our setting.

2.3 Screen Sharing

Advertisements are usually displayed on either a part or the whole user screen in an advertising supported app. For example, the banner advertisement of Google Play Services Ads is shown through an AdView component in the host activity's layout, while interstitial advertisements occupy the full screen of the imported AdActivity component. Though this work separates the advertising components from the original app to enhance privacy protection, the host and the advertising apps still need to share the user screen to maintain the original view. In this work, we take advantage of the Android WindowManager component to show a non-full-screen advertisement in a floating view. Full screen advertisement will not be influenced by app isolation for its standalone Activity layout. With our approach, we can seamlessly reconstruct the original app view even though app and library code reside in different application sandboxes.

2.4 Lifecycle Management

When decoupling their code base, we need to rewire the lifecycle of both host app and library. In particular, this means whenever the app is about to display an advertisement, the corresponding advertisement needs to be available and ready. This is handled through the IPC service connection and the floating window components.

2.5 Instruction Injection Tool

The novel Android Runtime features an on-device compiler that compiles dex bytecode to native code. The Optimizing backend, default since Android 6, first translates app code to its intermediate representation (IR) in the form of a per-method control flow graph. Afterwards, optimizations are passed and eventually code generation is executed on those graphs. ARTist complements the compiler with additional instrumentation routines, allowing for app modification on the IR level. In this work, ARTist is utilized to split app

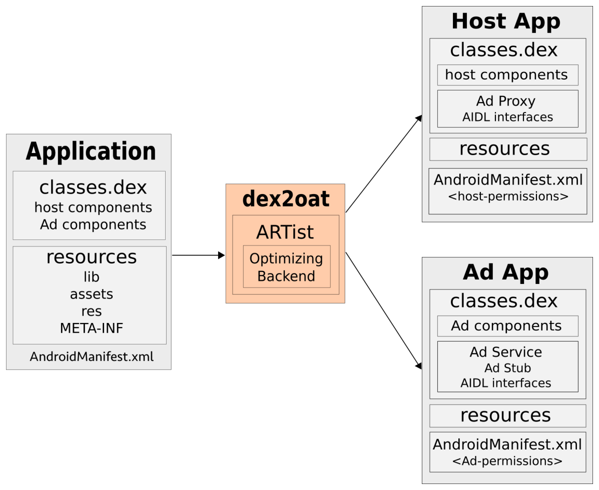

Figure 2: Architecture of ART-based App Compartmentalization Mechanism.

and library code and replace their communication channels through IPC-based ones.

2.6 Architecture

Our approach can be divided into two parts: application split and communication establishment. Figure 2 gives an overview on the architecture of our system.

Application Split. Since the compiler has full control over the app's bytecode and the fragmentation of advertising libraries is small, it is not difficult for compiler to distinguish library components through namespace of the target library and split them into different apps.

Communication Establishment. To establish communication between the two sub-apps, ARTist scans for broken connections between app and library and replaces them by invocations of AIDL-generated IPC stubs and proxies, which are injected into the corresponding sub-apps.

2.7 Performance

We distinguish between compile-time and runtime overhead. First, at compile time, splitting the application, installing the library and fixing their connection channels imposes additional overhead perceivable by the user in the form of a prolonged installation process. However, execution of our system is triggered by the user explicitly and only required once, so we argue that an overhead at this point is acceptable. Second, our approach induces the overhead of one additional IPC round-trip per invoked library function. However, this is infrequently used for first-time displaying of advertisements or lifecycle callbacks, so we do not expect the overhead to be perceivable by the user. And the selectable ahead-of-time recompilation here will not introduce additional startup overhead.

2.8 Deployment

Because we leverage the ARTist instrumentation framework to build our mechanism, we can find the detailed deployment process in [1]. Our final implementation is a normal app. In other words, our implementation is an application-level wrapping of the customized Optimizing backend. To construct the compilation environment in this app, all of the necessary libraries and executable files are imported into it, including a specialized version of the compiler. In order to compartmentalize an installed advertising-supported app, the only thing that users need to do is activating the compartmentalization mechanism by some simple clicks. After compilation, the two apps are installed, one of which is the app the user wanted to be separated from its advertisement code and the other is the advertising service app.

3. CONCLUSIONS

To remedy permission abuse and strengthen application resources protection in advertising-supported apps without system modification and application rebuilding, an ongoing compiler-based app compartmentalization mechanism is introduced in this poster. By customizing the ahead-of-time ART compiler, advertising components could be isolated from the host app and run as a standalone app. The extended instrumentation framework of ARTist helps establishing the communication between the host and the advertising app by instruction injection in the process of compilation. In this mechanism, the permission sets and resources of the separated sub-apps will be put into different sandboxes, forming a system-level trust boundary and providing an enhanced privacy protection with no need for OS customization and application rewriting.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) via funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345, 16KIS0656).

4. REFERENCES

[1] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. *arXiv preprint arXiv:1607.06619*, 2016.

[2] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. Free for all! assessing user data exposure to advertising libraries on android. *NDSS '16*, 2016.

[3] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. Hybrid user-level sandboxing of third-party android apps. In *ASIACCS '15*, pages 19–30. ACM, 2015.

[4] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. Flexdroid: Enforcing in-app privilege separation in android. *NDSS '16*, 2016.

[5] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *WiSec '14*, pages 165–176. ACM, 2014.

[6] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security '12*, pages 553–567, 2012.

[7] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *ASIACCS '12*, pages 71–72. ACM, 2012.