Poster: Full Support for Reference Monitoring in Android's Application Framework

Michael Backes
CISPA, Saarland University & MPI-SWS
backes@mpi-sws.org

Sven Bugiel
CISPA, Saarland University
bugiel@cs.uni-saarland.de

***Abstract*—In this poster, we present ongoing work on how to enable full support for reference monitoring in Android's application framework. By default, core services of the application framework are designed to accept data objects as input and to return data objects to application processes (e.g., Intent objects or clipboard data) instead of only references to these accessed objects. This design decision conceptually impedes, for example, the extension of SELinux type enforcement from Android's Linux kernel into Android's middleware, since type enforcement relies on persistently and reliably assigning security contexts to objects. When releasing data objects to applications, the integrity of the objects' security contexts cannot be ascertained anymore. By retrofitting Android's application framework to provide support for monitoring *references* to data objects (such as Intents or clipboards), we allow type enforcement to be efficiently extended to Android's middleware services.**

I. Motivation

Extensive application frameworks are more and more common in the software stacks of modern computing platforms, most notably on mobile devices like smartphones, but also increasingly on desktop machines. These frameworks offer to application developers feature-rich and high-level APIs to the onboard hardware, services, and data stores (e.g., location sensing, telephony, or contacts management). In addition to providing a very homogeneous execution environment, the application frameworks form an important aspect of application sandboxing to defend against nosy or malicious apps, for example, by restricting the privileges of each installed app.

Application frameworks have also been an active topic of security research over the last few years, with a very strong focus on Google's Android OS. As a reaction to various identified shortcomings of Android's security and privacy protecting mechanisms, Google continuously extended Android's security concept. More recently, Google integrated SELinux-based type enforcement into the Android Linux kernel and core system services [1] to harden the system against (the effects) of root exploits and to re-enforce the application sandboxing with mandatory access control. Also the research community has proposed various security extensions to the Android software stack. Among the proposed approaches for further enhancing Android's security architecture, the ones aligned most with Android's recent changes are aiming at extending type enforcement into Android's application framework in order to establish a more flexible and policy-driven protection of the end-user's privacy [2].

Established Reference Monitoring Systems: Security models such as type enforcement rely on securely assigning a security context to all subjects and objects in the system. Moreover, reliable reference monitoring requires that any operation between the subject and an accessed object is intercepted and submitted to an access control check. In established access control frameworks of operating system kernels, such as *Linux Security Modules* (LSM) [3]—which also forms the foundation for Android's SELinux support—or *Windows Kernel Objects* [4], these important requirements for reference monitoring are fulfilled by the design of the kernel API (i.e., syscall API) and the abstraction layers of the kernel objects. For instance, application processes access files, sockets, or other resources only through *handles* (or *references*), which are assigned by the kernel. Since any operation by the application has to resolve this reference, the kernel (and hence also LSM on Linux) can intercept and control any operation. Moreover, since the kernel objects are stored in the privileged context of the kernel, this enables securely assigning attributes, such as a security context, to the objects (e.g., through extended file-system attributes).

Incomplete support for reference monitoring in Android's application framework: Android's application framework design, however, conceptually fails in providing full *reference* monitoring support. This shortcoming impedes, for example, the extension of type enforcement from Android's Linux kernel to its application framework. A number of framework APIs return actual data objects to their callers instead of releasing only a reference to the accessed object. For instance, the *Intent subsystem* at the heart of inter-application communication on Android releases the actually sent Intent object to receivers and Intent life-cycles are generally managed within the application sandboxes. Thus, Intent objects cannot be securely assigned a persistent security context that governs access by different application processes to this object. Instead, an ephemeral context must be derived for an ad-hoc access control check whenever Intents are sent or received. Similarly, system services such as the clipboard service accept as input and return clipboard data objects instead of references to the clipboard data, hence impeding assigning a persistent, fine-grained security context to the clipboard.

II. Full Support for Reference Monitoring in Android's Application Framework

In this poster we present our ongoing work on retrofitting core services of Android's application framework with full support for *reference* monitoring of the managed data objects. Although this entails fundamental changes in the workflow of the affected services, a particular consideration of our

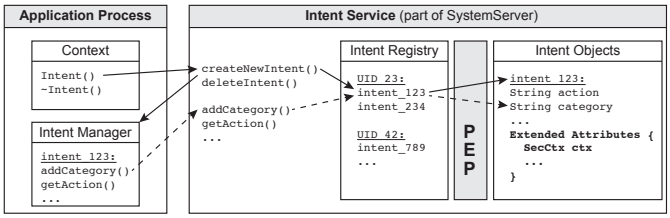

Figure 1. *Intent Manager* and *Intent Service* to enable reference monitoring of Intent objects. Solid line represents the work-flow for creating a new Intent object. Dashed line represents the work-flow for accessing the Intent object's fields and methods.

work is to retain the stability and backwards-compatibility of the framework API. To address this challenge, we exploit Android's concept of *Manager* and *Service* classes [5]. *Managers* are part of the Android SDK and encapsulate *Proxies* for system services (like location service or activity manager service). *Proxies* are a basic abstraction of Binder IPC between different application/service processes and implement a simple protocol for Binder-based remote procedure calls between those applications and services. To the application developer, *Managers* are simply local Java objects, which forward method calls via their encapsulated *Proxy* to their associated service. For instance, calling a function on the *LocationManager* results in an RPC to the *LocationManagerService*.

In our work we introduce new *Manager* classes that replace the framework SDK classes, which currently represent data objects that are released by the application framework services to application processes. Thus, to the application, our new *Managers* provide the same interfaces as the original data object classes and hence retain the stability and backwards-compatibility of the default Android API. However, our *Managers* do not contain any data (e.g., Intent data), but represent references (or handles) to the actual data objects maintained by a new, dedicated *Service*. To better illustrate this concept, we explain in the following our new *Intent Manager* and *Intent Service* as a replacement for the default Intent class. Together, they provide a central, secure management of Intent objects and a solution to releasing only references of Intent objects to application processes.

Intent Service and Intent Manager Example: Figure 1 illustrates the basic mechanics of our new *Intent Manager* and *Intent Service*. When an application process wants to create a new local Intent object (work-flow with solid line in Figure 1), the application's context will request the creation of a new Intent object from the *Intent Service*. The *Intent Service* then creates a new Intent object based on the original Intent class of the default Android application framework and assigns a unique ID to this new Intent object. Additionally, it creates a local registry entry that indicates that this Intent object is owned by the calling application process, using the application's UID as identity. The registry also counts the references by applications to this particular Intent object to allow the *Intent Service* to mark orphaned objects for garbage collection. The service returns the Intent's ID to the calling application, where a new *Intent Manager* object is created that internally stores the Intent's ID. To avoid memory leaks, the *Intent Manager* object informs as part of its Java object's finalization the *Intent Service* about its garbage collection, hence enabling the service to decrement the reference count on the associated Intent object.

For the application process, the *Intent Manager* object looks like a standard Intent object, which provides the same API as default Intents. For instance, the application process can call methods to add a category to the Intent (work-flow with dashed line in Figure 1). The *Intent Manager* forwards the method call together with the Intent's UID to the *Intent Service*, which subsequently performs the requested operation on the actual Intent object.

To ensure that a calling application process is actually authorized to perform a certain operation on an Intent object, any access has to pass a policy enforcement point (PEP) within the *Intent Service*. At the very least, the PEP ensures that an application process cannot access an Intent object that it does not own (as indicated in the Intent registry). However, more sophisticated policies, e.g., Intent firewalls or read-only Intent objects, can be centrally enforced at this PEP.

A particular benefit of this design is that the Intent objects are now managed securely in the context of the *Intent Service* process. This allows the implementation of extended attributes for Intent objects, such as security contexts, which are required for example for type enforcement. Since these attributes are internal to the *Intent Service*, this retains full compatibility with the standard Android SDK.

III. Conclusion and Next Steps

Our current implementation of supporting reference monitoring in the Android application framework already allows assigning persistent and secure attributes to monitored objects like Intents or clipboards. Our work-in-progress deals with the integration of the new *Manager* classes into other services of the application framework. For instance, sending an Intent is in our framework reduced to forwarding a reference to the Intent object. This entails different challenges (such as multiplexing access to the same object) and interesting aspects for the amortized performance overhead (e.g., sending references is more size-efficient than full objects, but every field access to an Intent object involves an IPC round-trip). Although our current implementation targets the Android OS, we consider similar use-cases for other mobile operating systems.

References

[1] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS '13)*. The Internet Society, 2013.

[2] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proc. 22nd USENIX Security Symposium (SEC '13)*. USENIX, 2013.

[3] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General security support for the Linux kernel," in *USENIX Security'02*. USENIX, 2002.

[4] Microsoft Corp, "Kernel objects (windows)," https://msdn.microsoft.com/en-us/library/windows/desktop/ms724485(v=vs.85).aspx.

[5] P. Brady, "2008 google i/o: Anatomy & physiology of an android," https://sites.google.com/site/io/anatomy--physiology-of-an-android, 2008.