Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

OS Support For Capabilities In Android

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Abdallah Dawoud

angefertigt unter der Leitung von / supervised by

Dr. Ing. Sven Bugiel

begutachtet von / reviewers

Dr. Ing. Sven Bugiel

Prof. Dr. Christian Rossow

03.01.2018

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, January 2018 Abdallah Dawoud

Erklrung

Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, January 2018 Abdallah Dawoud

Abstract

Android's security model utilizes a combination of low-level and high-level security mechanisms, such as the user-based protection model, SELinux, and permission system, to control access to system resources. However, this model has two limitations: First, it does not apply the principle of least privilege (PoLP) among app's components and, second, it fails short in tracking transitive invocations. The first limitation introduces the problem of malicious 3^{rd}-party libraries, whereas the second limitation enables the confused deputy attacks.

To address the problems caused by both limitations, we extended Android's security model with new security features borrowed from capability-based security model. Specifically, we introduced capabilities into Android's middleware with kernel support. The goal is to come up with a functional prototype that enables different components of the same app to run with different access rights on the high-level system services, respecting the PoLP. Additionally, the prototype must provide a clear path to mitigate confused deputy attacks targeting system services through channels that have deliberately exposed by the deputies.

Along the line, we use the Binder framework, which is used for IPC in Android, as the building block for creating and communicating capabilities of system services. We also rely on kernel's security guarantees to prevent forging capabilities. Additionally, we employ Android's permission model to reflect the dynamic high-level security decisions made by end users in order to encode the correct access rights into issued capabilities. As a result, we fulfill our goal without significantly increasing the attack surface or causing a performance degrade. In fact, our design shows a performance gain in specific places.

Acknowledgements

This master thesis would not have been possible without the support of many people. So, I would like to pay special thankfulness, gratitude, and appreciation to the persons below who made my research successful and assisted me, personally and professionally, at every point to achieve the goal:

My supervisor Dr. Ing. Sven Bugiel, who has always been very supportive and welcoming. Thank you for your valuable insights and discussions during this amazing experience. I feel fortunate to work with you.

Prof. Dr. Christian Rossow for accepting to review this thesis. Thank you.

My colleague and officemate Dhiman Chakraborty, who helped me a lot when I started working on this project. Thank you for being so friendly and supportive.

My colleagues and friends Jonas Cirotzki and Wadah Al-Hamadi, with whom with I had very interesting discussions.

My sisters, brothers, sisters-in-law, brothers-in-law for being there when I needed you.

My wife for her love, encouragement, and the unconditional support during the past four years. Thank you Eslam for being so understanding. I will be grateful forever for your love. Also, thank you my little kids Layan and Amro, for being the hugest source of happiness to me.

Finally and for most, Mom and Dad, whom I miss so much. Thank you for your unconditional support and love. Thank you for your prayers and encouragement. Any accomplishment I ever had, or I will have, will be credited back to you.

Contents

Abstract **vii**

Acknowledgements **ix**

Contents **xi**

List of Figures **xiii**

Listings **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Structure 2

2 Technical Background and Problem Statement **3**

2.1 Android platform 3

2.1.1 System Services 6

2.2 Android's Security Model 8

2.2.1 Application Sandboxing 8

2.2.1.1 User-Based Model 9

2.2.1.2 Security-Enhanced Linux 11

2.2.2 Android's Permission System 13

2.3 Binder Framework 18

2.3.1 Context Manager 25

2.4 Problem Statement 29

3 Related Work **31**

3.1 Object-Capabilities 31

3.2 Capsicum: Capabilities in UNIX Systems 32

4 Design and Implementation **35**
4.1 Design Decisions . . . 36
4.2 Big Picture . . . 37
4.3 Management of Capabilities . . . 42
4.3.1 Capabilitys Access Rights . . . 42
4.3.2 Acquiring Capabilities . . . 45
4.3.3 Capability Invocation . . . 46
4.3.4 Capability Delegation . . . 48
4.3.5 Revocation of Capabilities . . . 51
4.4 Implementation . . . 52
4.4.1 Scope and Limitations . . . 52
4.4.2 Changes on AOSP and Kernel . . . 54

5 Security Analysis **57**
5.1 Assumptions . . . 57
5.2 Attacker Model . . . 58
5.3 Attack Scenarios . . . 59
5.3.1 Mitigated Attacks . . . 59
5.3.1.1 Confused deputy . . . 59
5.3.1.2 Inclusion of Malicious Library . . . 60
5.3.2 Attacks Against Our Design . . . 62
5.3.2.1 Collude attacks . . . 62
5.3.2.2 Overwriting Access Rights . . . 63

6 Discussion and Evaluation **65**
6.1 Performance Analysis . . . 65
6.1.1 Configurations and Setup . . . 65
6.1.2 Experiments and Results . . . 66
6.2 Coverage and Effectiveness . . . 69
6.3 SELinux vs. Capabilities . . . 72

7 Future Work **77**

8 Conclusion **81**

Bibliography **83**

List of Figures

2.1 Android's Software Stack 4
2.2 Direct/Indirect Access of Kernel Resources 6
2.3 Global Access Control Enforcement by DAC, MAC, and Permissions [29] 13
2.4 Granting And Revoking Permissions 14
2.5 Binder Framework 19
2.6 High-Level View on Proxy and Stub Classes 24
2.7 Registering System Services 27

3.1 Capabilities and Objects 32

4.1 Abstract Flows For Acquiring, Invoking, and Delegating Capabilities . . . 40
4.2 Reporting Permissions 44
4.3 Requesting Capabilities 47
4.4 Transferring Binder Handles Between Apps 49

5.1 Mitigating The Confused Deputy Attack 60
5.2 Mitigating The Problem of 3^{rd}-party Library 61

6.1 Time Measurement Components 66

Listings

2.1 File's UID and GID 10
2.2 Process Status Information 10
2.3 SELinux Rules 12
2.4 Reference Monitor In Wi-Fi Service 17
2.5 Transaction Data 21
2.6 Portions Of Binder Node Structure 22
2.7 Portions of Binder Reference Structure 23
2.8 Portions of Binder Process Structure 23
4.1 Reference Monitor On Capabilities 48
4.2 Reflection To Extract Binder Handle From Manager And Constructing The Manager Again 50
6.1 SELinux Allow Rules 73

List of Tables

2.1 Permissions Enforced By DAC and MAC 18
2.2 Permissions That Protect Sensitive Broadcast Messages 18

4.1 Permissions Required Per Service 45
4.2 LoCs Introduced By Our Design 54
4.3 Newly Introduced Methods 55

6.1 AOSP and Kernel Build Information 66
6.2 Permissions Can Be Enforced By Capabilities 71
6.3 Permissions That Is Not Supported By Our Design 72

Chapter 1

Introduction

1.1 Motivation

As Android devices are rapidly evolving in power and capacity, they are gradually becoming an indispensable element in our lives. As of May 2017, Android has reached 2 billion monthly active devices globally [1]. Naturally, an open platform, with such a huge population, would attract malicious developers who would search every corner for an exploit to launch attacks endangering security and privacy of this great mass.

The current security model of Android aggravates the problem as it fails short to apply the principle of least privilege among app's components. In other words, each component running inside an app, would have the exact same privilege as any other. This opens the door for the infamous problem of malicious 3^{rd}-party libraries which announce specific functionalities (and, thus, get included in apps) but perform other undeclared malicious operations misusing the power given to them.

Another shortcoming of the current security model of Android is its inability to track transitive invocations. In other words, the system would serve requests based on the identity of the last caller. This disregards the fact that the request might have been originated by an entity other than the last caller, which might not have the required privilege to access the resource. This opens the door for the confused deputy and collude attacks.

For both limitations, researchers have been introducing several novel techniques that vary from being application-level or platform-level solutions. However, we believe we can contribute to the solution by extending the current security model with new security concepts which are borrowed from another security model, namely, object capabilities.

1.2 Thesis Structure

This thesis is structured as follows. In *Chapter 2 Technical Background and Problem Statement*, we start by introducing Android's architecture and the security features contributing to its security model. Then we elaborate on the Binder framework which represents the building block of our design. We end the chapter by the problem statement that illustrates, in details, the problem we are trying to solve in this work. In *Chapter 3 Related Work*, we present a brief introduction about capabilities and then present Capsicum project, which brings capabilities into UNIX systems. In *Chapter 4 Design and Implementation*, we present the details of our design and discuss its limitations. We also show some stats on type and amount of changes introduced by our implementation. In *Chapter 5 Security Analysis*, we discuss how we can employ the security features provided by our design in leveraging system's security. In *Chapter 6 Discussion and Evaluation*, we evaluate our design against stock and discuss whether it is possible for SELinux provide the same advantages of capabilities. In *Chapter 7 Future Work*, we present our ideas on how to improve our design and address about open issues. We finally conclude the thesis in Chapter 7 Conclusion.

Chapter 2

Technical Background and Problem Statement

In this chapter, we present the technical background that paves the way for introducing our design. We begin by presenting Android's platform and discussing the cornerstones of Android's security model. Then, we elaborate, in details, on the Binder framework, which forms the building block of our design. Based on the presented knowledge, we end the chapter by stating the problem we are addressing in this work.

2.1 Android platform

Figure 2.1 shows a high-level view of Android platform [6], which is also known as Android's software stack. At the very bottom of the stack, we find a modified Linux kernel which, in addition to the conventional functionalities of any UNIX kernel, provides special drivers for GPS, camera, audio, Bluetooth, graphics, among others. The most important driver for this work is the Binder, which enables an efficient inter-process communication (IPC).

On top of the kernel, there is the Hardware Abstraction Layer (HAL). This layer acts as a bridge between kernel space and user space. HAL provides interfaces for the kernel drivers, called HAL modules, which must be implemented by hardware vendors. The contract on how these interfaces are invoked by user space processes is well-defined. However, the underlying implementation could differ from one device to another.

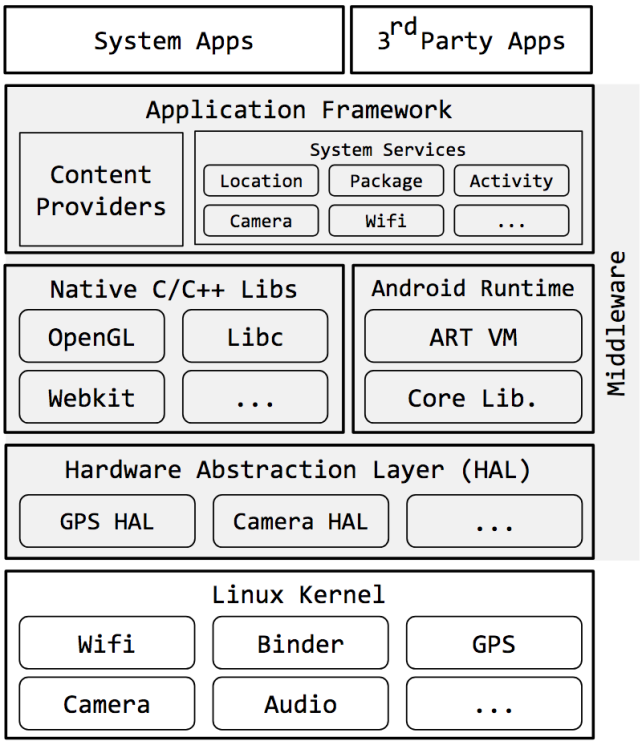


FIGURE 2.1: Android's Software Stack

In Android, each app runs in its own process with a private instance of Android Runtime (ART). The ART is responsible for translating the Dalvik Executable (DEX)[1] bytecode into machine code. DEX files result from compiling Android apps (which are written in Java) using build toolchains, such as Jack [10]. Android Runtime includes Java and Android core libraries, such as packages of *java.**, *javax.**, and *android.**. These libraries provide most of the functionalities of the Java programming language (such as string handling, files management, and networking) in addition to the libraries that are specific for Android development (such as the libraries used for building interfaces, accessing hardware, and sharing data among apps).

Next, we see the native C and C++ libraries which are used heavily by HAL modules and ART. Some functionalities of these native libraries are exposed to the application framework using Java Native Interface (JNI). Examples of these functionalities are: Creating internet sockets, accessing files [19], storing persistent data in relational SQLite databases, compressing data, and rendering 2D/3D graphics.

The application framework provides APIs to a wide range of system services. Developers rely on these services to enrich their apps with complex functionalities, such as access to location, camera, sensors, and telephony services. The application framework also provides APIs that enable apps to access data stored by the system (such as contacts, and

[1] In Android 5.0, ART came as a replacement for Dalvik virtual machine. Nevertheless, the abbreviation DEX still refers to Dalvik.

calendar entries) and data stored by other apps as a form of data sharing between apps via what is called "Content Providers". It worth noting that the application framework implements a considerable amount of its functionalities in native code and accesses it through JNI.

At the very top of Android's software stack, we find system and 3^{rd}-party apps. Examples of system apps are the Settings (for modifying and viewing settings of system and installed apps, changing permissions of apps, among other functionalities), Contacts (which uses the contacts provider to perform CRUD operations on the contacts address book), and Dialer (for making and receiving phone calls). Users can install 3^{rd}-party apps from Google Play and arbitrary other sources. Each app has a manifest file called *AndroidManifest.xml* that defines app's behavior and capabilities. App developers can compile C/C++ code into static libraries and include them into app's package, e.g., using Android's Native Development Kit (NDK). As mentioned earlier, these libraries can then be accessed from Java code using JNI.

Android apps are composed of four basic components, which are:

- *Activities*: The user interface which users can interact with.
- *Services*: Components that execute long-running operations in the background. A service provides the means for other apps to bind with it in order to use its exported methods.
- *Content Providers*: Implement a mechanism for apps to share data among each other using SQLite-like interfaces.
- *Broadcast Receivers*: Resemble mailboxes for broadcast Intent[2] messages.

It worth noting that all app processes are forked from a process called Zygote [5]. Zygote is a pre-warmed process that includes common framework code and resources but not the operational code of the apps. After Zygote forks an app's process, it loads app's code into the process and starts it. As a consequence, all processes forked from Zygote share the memory blocks allocated for framework code and resources. This technique is meant for speeding up the launch time of apps and optimizing memory usage.

Finally, we should mention that the complete Android's software stack resides in two projects: Android Open Source Project (AOSP) [7] and the Goldfish kernel project [17]. **For all discussions and code tracing throughout this thesis, we use AOSP of version 7.1.2_r33 (code name Nougat or N for short) and Goldfish of version 3.4 as references (unless stated otherwise).**

[2] An intent describes an operation to be performed against components of self and other apps. For example, it is used to start activities, send broadcasts, and start or bind with services.

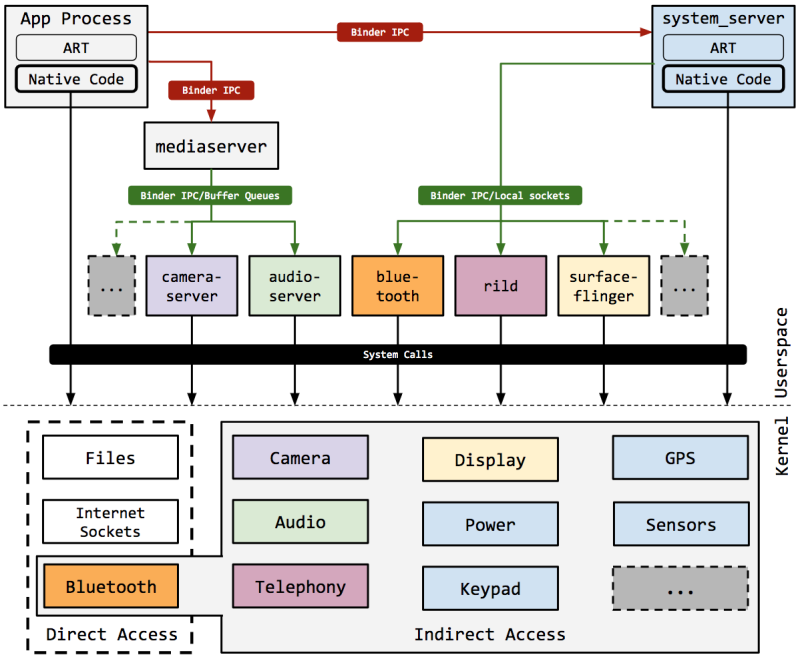


FIGURE 2.2: Direct/Indirect Access of Kernel Resources

2.1.1 System Services

As shown in Figure 2.2, Android provides several kernel resources that can be accessed by Android apps to perform specific functionalities, such as retrieving GPS coordinates, downloading data from the internet, and capturing audio and video. Based on the form of access, these resources are divided into two main categories:

- **Resources Accessed Directly**: Android apps can access this type of resources either by issuing direct system calls from app's native code or by invoking high-level APIs from the application framework[3]. In either case, it is the app's process that issues the system calls. Examples of this type of resources are files, internet sockets, and Bluetooth sockets[4] [21][14][15].

- **Resources Accessed Indirectly**: Android comes with two highly-privileged processes called *system_server* and *mediaserver* [9]. Both processes expose some of the low-level functionalities to be invoked by apps over IPC. Similar functionalities are wrapped in what is called a "*System Service*". Although the *system_server*

[3]The application framework uses native code of its own to issue system calls to the corresponding kernel modules. For example, an app uses the *File.java* from the application framework to open a file, referenced by its full path. The call passes through different classes and layers until it eventually reaches *libcore_io_Posix.cpp* [19] which issues a system call that opens a file and returns its file descriptor. This native library can also be used to create IPv6 and IPv4 sockets.

[4]There have been some efforts in bypassing the official Bluetooth APIs provided by the application framework to access Bluetooth functionalities from native code. Supported by the fact that Android maps Bluetooth permissions into groups (more on this later in this chapter), we believe this resource falls into both categories and therefore, can be accessed directly and indirectly.

manages its own drivers (see the drivers colored with the light blue in the Figure 2.2), both *system_server* and *mediaserver* processes use another level of indirection to have access to more kernel drivers. For example, the *mediaserver* instructs the *cameraserver* process to obtain video frames from the camera device (through an HAL module) and return the frames in a buffer queue [11]. Moreover, the *system_server* communicates with *rild* (Radio Interface Layer daemon) over local sockets to query information about the telephony services, register listeners for changes on phone state, among other functionalities. The introduction of system services reduces the programming effort on app developers who can access the low-level functionalities through high-level APIs from the Java code.

By introducing system services, the kernel has to authorize a bounded number of processes to access its resources in comparison to what would have been the case if each app's process would access the kernel resources itself. This significantly reduces the management overhead caused by the enforcement of access control in the kernel. However, to preserve stability of the system, the *system_server*, *mediaserver*, and other processes that host system services have to authorize apps which invoke their functionalities, resembling the access control as it would have been enforced by the kernel. In turn, the kernel would authorize these processes. Although most of the access control logic is moved to the middleware, the kernel would still have to authorize apps when they attempt to access the resources exposed directly to them, such as accesses to files and internet sockets.

In Android, the list of system services gets longer with each release[5]. As of Android 7.1.2, the number of all system services is about 115[6]. However, about 83 of the total number of system services are exposed to app developers [4] while the rest is meant to be used internally by the Android framework.

App developers can access most system services in a unified fashion. Specifically, developers use the *getSystemService(String name)* API from app's context [8] to retrieve a manager for the remote system service that is referenced by the name supplied to the call. Developers then invoke methods of the manager. In turn, the manager passes the request to the remote system service over IPC. Based on the sensitivity of invoked method, the system service checks authorization of the caller process, executes the desired method if the request is legit (e.g., process is authorized and parameters are as expected), and return a result, if any.

[5]It worth noting that different Android vendors can also add more system services than what exists in the AOSP.

[6]This information was retrieved by running the shell command: *adb shell service list* against the default Android emulator of AOSP. It worth noting that some services cannot be emulated, e.g., the bluetooth service [20], and therefore they do not appear in the service list.

Examples of the services that can be retrieved by 3^{rd}-party apps are: ActivityManagerService (AMS), LocationManagerService (LMS), WifiService, CameraService, PowerManagerService, and WindowManagerService. Some system services, such as the PackageManagerService (PMS), require to be bound to the app's context and, therefore, they should be retrieved using other API, namely, *getPackageManager()* from app's context.

Android enables app developers to create their own services and expose them to other apps. Those services are called bounded services. Although both system and bounded services use IPC, the mechanics on how apps connect to bounded services (from middleware perspective) is different from how apps retrieve handles (connect) to system services. What makes system services really special is the fact that they are registered with a component called the Context Manager (process name is *servicemanager*). As we will discuss later in *Subsection 2.3.1 Context Manager*, this technique is used to expose system services to apps and other middleware processes.

2.2 Android's Security Model

There are several technologies that contribute to the security model of Android, some of which are directly inherited from the Linux kernel (e.g., user-based model, SELinux[7], virtual memory, ASLR, and DEP) while others have been introduced and adapted specifically for Android (e.g., Android's permission system and Binder framework for IPC). All of these technologies aim to provide storage and memory isolation, mitigate software exploitations, control access to resources, allow for secure and efficient IPC, among other goals.

In this section, we limit the discussion to application sandboxing, a technique that is used to limit the capabilities of the environment in which a process is executing, and Android's permission system, that is used to control access beyond boundaries of sandboxes. We dedicate *Section 2.3: Binder Framework* for a detailed discussion on the Binder framework for its importance to our work.

2.2.1 Application Sandboxing

In Android, users install arbitrary 3^{rd}-party apps which might contain bugs, security vulnerabilities, and malware. To prevent malicious apps from sabotaging other apps and the host system, Android utilizes the user-based protection model to lay the ground rules

[7]The user-based model and SELinux enforce Discretionary Access Control (DAC) and Mandatory Access Control (MAC), accordingly.

for application sandboxing and uses SELinux to further limit the capabilities of each sandbox. As a result, each app would run in a restricted environment with a limited set of resources and confined access rights on these resources.

Although it is called "application" sandboxing, the techniques discussed here are also applied to all processes in the system, including middleware processes (*system_server* and *mediaserver*), daemons (*rild*), and some other highly-privileged processes, such as *Zygote*. This is crucial to mitigate attacks targeting the system through these processes.

2.2.1.1 User-Based Model

Traditional Linux systems rely on the user-based protection model, which enforces Discretionary Access Control (DAC), to support multiple users, where each physical user is assigned a unique user ID (UID). Such systems assume that different users should not trust each other. Therefore, each UID is allocated a subset of the resources in the system. Processes that run under the same UID share the same resources assigned to it but cannot, by default, access resources allocated to other users. Android takes advantage of this model and implements sandboxing, unconventionally, by assigning each app a unique UID at app's installation time. As a result, processes of different apps would run with isolation of each other on two levels:

- **Resource Isolation**: Each app has its own private storage that can be accessed only by its processes. Moreover, processes are prevented from accessing most of the system services and kernel resources. For example, an app's process cannot, by default, create internet sockets nor retrieve coordinates from the location service.
- **Memory Isolation**: When *Zygote* forks an app process, it assigns it the UID of the owning sandbox. This guarantees memory protection for all app processes because two processes of different sandboxes would be assigned unique UIDs.

It worth noting that Android supports multi-user feature where the user in this context is called a profile. However, this feature is disabled, by default, and has to be enabled by device manufacturers [12][46].

Android's implementation of sandboxing allows multiple processes of the same app to run under one sandbox[8]. Hence, they have access to the same resources. When a sandboxed process fails or gets compromised, other processes running in the same sandbox would be affected as they share the same resources. However, the effect remains contained and does not propagate to outside processes.

[8]Two processes of different apps can share the same sandbox if their apps were signed by the same signature [2], whereas processes of the same app run, by default, in the same sandbox.

Following our discussion in *Subsection 2.1.1 System Services*, the kernel resources that can directly be accessed by apps are assigned to app sandboxes using standard Linux facilities, namely, the UID and group ID (GID). Examples of these resources are the virtual file system and network sockets. To better understand how the kernel control access to such resources, we discuss how an app is assigned, by default, a private directory with read and write access rights on it.

When an app is installed, the package manager service instructs the *installd* daemon over sockets [18] to create a directory for the newly installed app. As a consequence, a directory with app's package name is create under the directory "/data/data/". Files stored in the new directory are private to the owner app. The Listing 2.1 shows the result of *stat* command on a file named *private_file* that is created and stored in a directory assigned to an app with a package name "*com.test.app*". We can observe that this file is owned by the user with UID of *10070* and its main group is also *10070* (line 4), which happens to be the UID of the newly installed app with the package name "*com.test.app*". The access rights allow the owner or processes that join the group *10070* to read and write the file (660 access rights, line 4). However, other processes which are not the owner nor has joined the specified group cannot perform any operation on the file.

When *Zygote* forks a process for the newly installed app, it sets the UID, main GID, and supplementary GIDs of the process. The Listing 2.2 shows the status information of app's main process. We can see that this process is, in fact, the owner of the file (line 4). Therefore, it can read and write it. We can also see that it joins two supplementary groups (line 7). The first group (*9997*) is given to all apps of the same profile while the second group (*50070*) is shared across all profiles for the same app. The supplementary groups reflects extra permissions assigned to the process. For example, if a process has the *3003* group, then it is allowed to create IPv4 and IPv6 internet sockets [21].

```
1 # stat /data/data/com.test.app/files/private_file
2   File: 'private_file'
3 {...}
4 Access: (660/-rw-rw----)        Uid: (10070/  u0_a70)   Gid: (10070/  u0_a70)
5 {...}
```

Listing 2.1: File's UID and GID

```
1 # cat /proc/{PID}/status
2 Name:   com.test.app
3 {...}
4 Uid:    10070   10070   10070   10070
5 Gid:    10070   10070   10070   10070
6 FDSize: 64
```

```
7 Groups: 9997 50070
8 {...}
```

Listing 2.2: Process Status Information

2.2.1.2 Security-Enhanced Linux

Application sandboxing using DAC sets the foundations for storage and memory isolation. However, DAC fails short to apply fine-granular permissions on resources. For example, a process can either execute all functionalities of file or none based on wether the process has the execute permission on the file or not, accordingly. Moreover, processes that run under the same UID enjoy the same access rights on the resources allocated to them. Thus, the problem of gaining full access to system's resources boils down to exploiting one vulnerable root process.

Integrating SELinux in Android enforces a Mandatory Access Control (MAC) policies that confine access to files and network resources [26]. SELinux overcomes the limitations of the DAC by defining policies for accessing resources at the level of a single operation. The enforcement of policies enables privilege segregation across processes running under the same UID. This means even root processes (processes that run under UID of *0*) are not equal in privilege [22]. In fact, the notion of users does not exist in SELinux. The absence of a policy is interpreted as an access denial. Therefore, explicit policies must be defined to cover all legit access scenarios. Using SELinux does not eliminate DAC. Instead, both mechanisms complement each other as SELinux policies are checked after DAC permits the access.

SELinux policies are static in the sense that when they are defined, they cannot be changed. In Android, SELinux policies are compiled and shipped as a single binary file that is used by the Linux Security Module (LSM) to regulate accesses to resources (SELinux is implemented as part of the LSM framework). Adding a new policy requires installing a new compiled policy file, which include the new policy, into the system. In other words, a new ROM needs to be installed on the device to apply new policies.

LSM Hooks

The LSM hooks of SELinux are special security functions which are placed in kernel and user space code where access control checks are needed. For example, LSM hooks reside in Android middleware (e.g., *service_manager.c* and *DrmManagerService.cpp*), the kernel-level Binder driver (*binder.c*), among other critical places where operations on files and socket, or accesses of resources of other processes are done. The LSM of SELinux uses the security contexts of both subject and object along with the action to decide whether to allow an operation or not (based on the defined policies).

Security Contexts

SELinux relies on the concepts of subjects, objects, and actions. Subjects are processes trying to perform actions on objects, whereas objects are resources managed by the kernel such as files, sockets, and processes. Each subject and object must belong to a security context. The security context is composed of four components: User, role, type, and level. These four components enable Role-Based Access Control (RBAC), Multi-Level Access Control (MLS), and Type-Based enforcement (TE). For this thesis, we only focus on TE as a mean for privilege confinement.

Type-Based Enforcement

Each subject and object must have a single type, which is defined in the associated security context. Types must be created statically by the system administrator. Nonetheless, the administrator can also define transition rules that define the new types of subjects and objects based on a specific event, namely, executing a file. Another way of assigning types is with forking as child processes inherit types of parents. Rules use types to define access rights on resources. LSM makes sure that only accesses with defined policies are permitted.

To better understand how SELinux is used to confine privileges of processes, we consider the rules listed in Listing 2.3. In the first line, we see a rule that constitutes that a process of type "system_server" can execute "*add*" and "*find*" functions from an object of type *system_server_service* and of class *service_manager*[9]. Based on the policy, invoking another function (e.g., *list*) will fail. An LSM hook that enforces the first rule would ideally be placed in the place where the *service_manager* receives a request from another process to add or find a service maintained by it. The LSM hook will retrieve the security contexts of both parties and consult an entity, namely, the Security Server, that can decide whether the request should be permitted or not. The second Line shows another policy that defines what operations the process *Zygote* (that has a type of *zygote*) can perform on a directory labeled with *zygote*. It worth noting that labeling and assigning classes to subjects and objects happen when the system boots up.

The more defined rules imply more fine-grained access rights. However, that would cause more overhead as policies are compiled into a single binary file. Although LSM implements caching of policies to enhance performance, the number of policies is preferred to be as minimal as possible.

```
1 allow system_server system_server_service:service_manager { add find };
2 allow zygote zygote : dir { ioctl read getattr lock search open } ;
```

[9] Android defines several object classes in *system/sepolicy/security_classes* and defines the possible operations on them in the file *system/sepolicy/access_vectors*. Examples of these classes are file, socket, dir, process, and service_manager. The former class defines three operations: add, get, and list.

LISTING 2.3: SELinux Rules

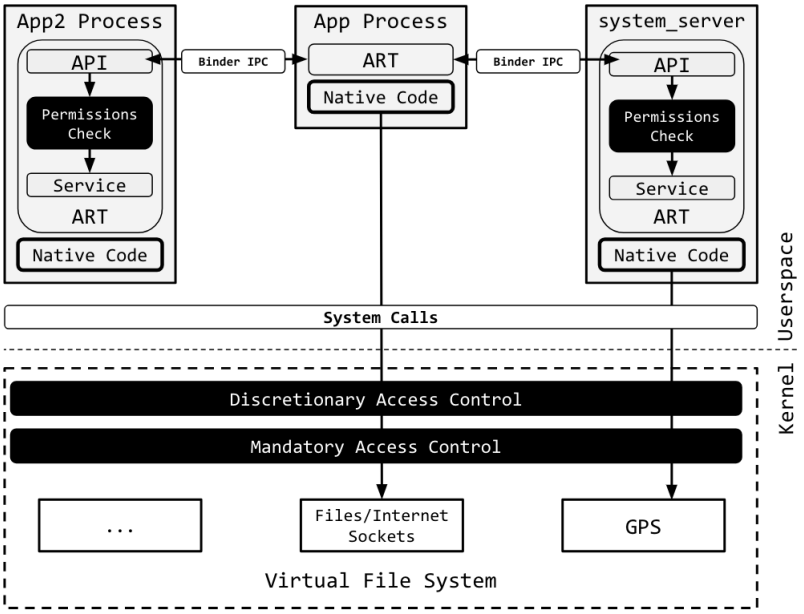


FIGURE 2.3: Global Access Control Enforcement by DAC, MAC, and Permissions [29]

2.2.2 Android's Permission System

As discussed in *Subsection 2.1.1 System Services*, Android has introduced system services which manage several kernel resources. Apps have no other option but to go through these system services to access the kernel resources managed by them. As a consequence, the kernel has to authorize system services, conventionally, using UIDs and GIDs permissions in addition to SELinux policies. In turn, the system services must authorize calling apps using Android's high-level permission system (see Figure 2.3). Along the line, the new permission system comes with extra benefits:

- It grants the system more control on what functionalities can be exposed or hidden to/from 3^{rd}-party apps because the access control happens in the middleware which has the high-level semantic that is missing in the kernel. For example, the system prevents 3^{rd}-party apps from adding new location providers while allowing system apps to do so.
- This design enables dynamic permissions in the sense that apps can be granted permissions while they are running without the need to restart them. This is one of the advantages over the POSIX permissions that are enforced by GIDs and UIDs which require the restart of the running app so the permissions become effective.

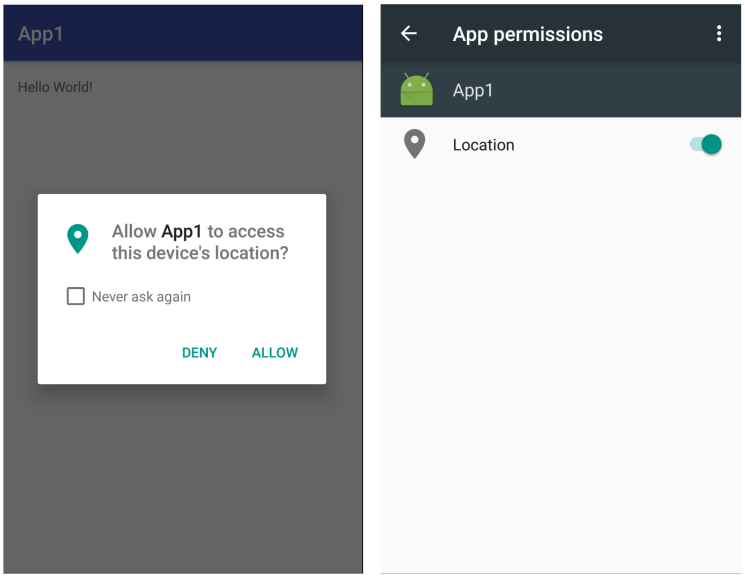


FIGURE 2.4: Granting And Revoking Permissions

- The system uses the permission system to protect data accessible through content providers. Such as contacts address book and calendar entries.
- It is generic enough to be used between apps. For example, an app can define a permission and protect a service it provides.

In the new permission model, system services protect their methods by permissions, which are unique strings that denote the ability to perform specific methods [33]. App developers have to request permissions in order to invoke the protected methods of system services. The decision on whether to grant, or revoke the permissions is placed in the hands of the end users. If the end user denied the request of granting permission, the app cannot trigger execution of the protected methods.

Permissions Management

Android keeps a database of all permissions granted or denied in the file */data/system/-packages.xml*. One of the many tasks provided by PMS is to maintain this file at runtime. Whenever a permission is granted or denied, the package manager reflects the change into this file. Permissions are stored in the database along with the package name and the corresponding UID of the sandbox.

The package manager provides APIs to query wether a specific sandbox has a specific permission or not. This is heavily used by system services to decide on whether to serve or reject the incoming invocation.

Requesting/Acquiring/Revoking Permissions

The app developer has to decide in prior what permissions are needed in her app so she would request them in the AndroidManifest.xml file. Based on the permission type, the end user will be asked either at the install time or at the runtime to grant the permission to the app. Figure 2.4(a) shows a typical permission dialog asking to grant (App1) the permission to access the location. Permissions can also be revoked using the Setting app, see Figure 2.4(b).

Permission Levels

System and user-defined permissions can take one of the following protection levels [23]:

- *Normal*: They are of low-risk to the system, apps, and end user.
- *Dangerous*: They are sensitive in terms of security and privacy because they grant the app access over user's private data and device which, if put in wrong hands, would cause severe impact on the user.
- *Signature*: This type of permissions is only granted if the requesting app is signed with the same signature used to sign the app that defined the permission.
- *signatureOrSystem*: This type of permissions is similar to the signature. However, it is also granted to applications that are in the Android system image.

Permission Types

Starting from Android 6 [24], permissions are divided into two types:

- *Install permissions*: This type of permission is granted at install time and cannot be revoked. Install-time permissions include normal and signature permissions.
- *Runtime permissions:* In general, dangerous permissions are granted at runtime. However, unlike install-time permissions, runtime permissions can be revoked using the Settings system app.

User-Defined Permissions

Although it is out of our scope in this thesis, it worth noting that Android enables app developers to define permissions in order to protect app's components. For example, the developer has the option to only allow the apps that have a specific user-defined permission to start an activity of her app, bind with a service her app offers, send a

broadcast which she creates a receiver for, or access a content provider her app maintains.

Permission Groups

Starting from Android 6 [24], all dangerous permissions belong to groups. These groups are: CALENDAR, CAMERA, CONTACTS, LOCATION, MICROPHONE, PHONE, SENSORS, SMS, and STORAGE. When an app requests a dangerous permission declared in its manifest file, the system handles the request in two different ways depending on whether the app is already granted a permission from the same group or not. First, if the app is not granted a permission from the same group, then the user is prompt with the permission dialog message. As shown in Figure 2.4(a), the permission group is stated rather than the permission itself. Second, if the app is granted a permission from the same group, the system grant the permission to the app silently without any interaction with the user. This means, app developers would still have to declare all dangerous permissions individually in the manifest file and end users grant and revoke permissions per group.

Permission Enforcement

For the sake of relevance, we only discuss the permissions that are used to protect functionalities of system services. This means we refrain from discussing the enforcement of custom permissions defined by app developers and permissions that protect content providers.

As mentioned earlier, each system service protects its sensitive methods with permissions. When a protected method is invoked, the service extracts the UID and PID of the incoming request (which can be retrieved using *Binder.getCallingUid()* and *Binder.getCallingPid()*, accordingly) and use them to query the PMS to decide if the caller process has the required permission to access the desired method. In turn, the PMS uses the UID of the caller process to locate its permissions and return *PERMISSION_GRANTED* if the required permission is granted or *PERMISSION_DENIED* if it is denied. The system service acts based on this information by either allowing method's execution (if permission is granted), or raising a *SecurityException* (if permission is denied).

Listing 2.4, shows how LMS protects the *getLastLocation()* method. First, the method gets the resolution level through *getCallerAllowedResolutionLevel()* which extracts caller's PID and UID (lines 11-12) from the request and then passes the control to *getAllowedResolutionLevel()* where the computation of the resolution level actually happens. The former method computes the resolution level after consulting the PMS

through service's context (lines 16-17). The resolution level is then injected in *checkResolutionLevelIsSufficientForProviderUse()* (line 5) which would raise a *SecurityException* if the caller does not hold the required resolution to access the required provider.

```
1  public Location getLastLocation(LocationRequest request, String packageName) {
2      {...}
3      int allowedResolutionLevel = getCallerAllowedResolutionLevel();
4      {...}
5      checkResolutionLevelIsSufficientForProviderUse(allowedResolutionLevel,
6                  request.getProvider());
7      {...}
8  }
9
10 private int getCallerAllowedResolutionLevel() {
11     return getAllowedResolutionLevel(Binder.getCallingPid(),
12                 Binder.getCallingUid());
13 }
14
15 private int getAllowedResolutionLevel(int pid, int uid) {
16     if (mContext.checkPermission(android.Manifest.permission.
17                         ACCESS_FINE_LOCATION, pid, uid) ==
18                         PackageManager.PERMISSION_GRANTED) {
19         return RESOLUTION_LEVEL_FINE;
20     }
21     {...}
22 }
```

Listing 2.4: Reference Monitor In Wi-Fi Service

Although most permissions are enforced in similar fashion to what we have discussed above, some permissions are enforced in other ways:

- As of Android 7.1.2, there are five permissions available to 3^{rd}-party apps which are mapped, when granted, to GIDs [14], and therefore enforced by DAC and MAC, as discussed in *Subsection 2.2.1 Application Sandboxing*. Those permissions are listed in Table 2.1. The first three permissions cannot be revoked once granted, while the last two can be revoked using the settings app but require app to be restarted, only if it was running [13].

- Another set of permissions, shown in Table 2.2, are enforced by the system when it delivers broadcast Intent messages to apps that define these permissions and implement broadcast receivers that matche specific actions. For example, the system allows apps that define the normal permission *RECEIVE_BOOT_COMPLETED* and implement a receiver with action of *android.intent.action.BOOT_COMPLETED* to receive a broadcast when the system finishes booting.

Table 2.1: Permissions Enforced By DAC and MAC

No.	Permission	Level
1	BLUETOOTH_ADMIN	normal
2	BLUETOOTH	normal
3	INTERNET	normal
4	READ_EXTERNAL_STORAGE	dangerous
5	WRITE_EXTERNAL_STORAGE	dangerous

Table 2.2: Permissions That Protect Sensitive Broadcast Messages

No.	Permission	Level
1	RECEIVE_BOOT_COMPLETED	normal
2	PROCESS_OUTGOING_CALLS	dangerous
3	RECEIVE_SMS	dangerous
4	RECEIVE_WAP_PUSH	dangerous
5	RECEIVE_MMS	dangerous

2.3 Binder Framework

The process isolation is a desired feature in any operating system. However, processes need to communicate, e.g., to exchange data and invoke operations from each other. Communication across processes, or IPC, holds a higher significance in Android than any other traditional system, considering the platform design choices, where services and components are decoupled from each other for modularity and security reasons. For example, unlike most applications running on traditional operating systems, the simplest action of switching between two activities in Android, even within the same app's process, requires an IPC to another process, e.g., to let the AMS (which resides on the *system_server* process) handle the request.

Since Android uses a Linux kernel, it ideally supports all traditional IPC mechanisms offered by the kernel, such as signals, shared memory, files, message queues, and sockets [33]. Some of these mechanisms are used in Android. For example, *system_server* uses local sockets to communicate with the radio daemon, the *mediaserver* uses buffer queues to receive video frames from the *cameraserver* [11], and apps can naively communicate through public files. Nonetheless, Android relies heavily on another IPC mechanism that is lightweight, secure, and enables synchronous remote[10] procedure calls (RPC); this mechanism is the *Binder*.

[10]The Binder does not support communications over the network and, therefore, "remote" in this context refers to services that run on other processes but within the system.

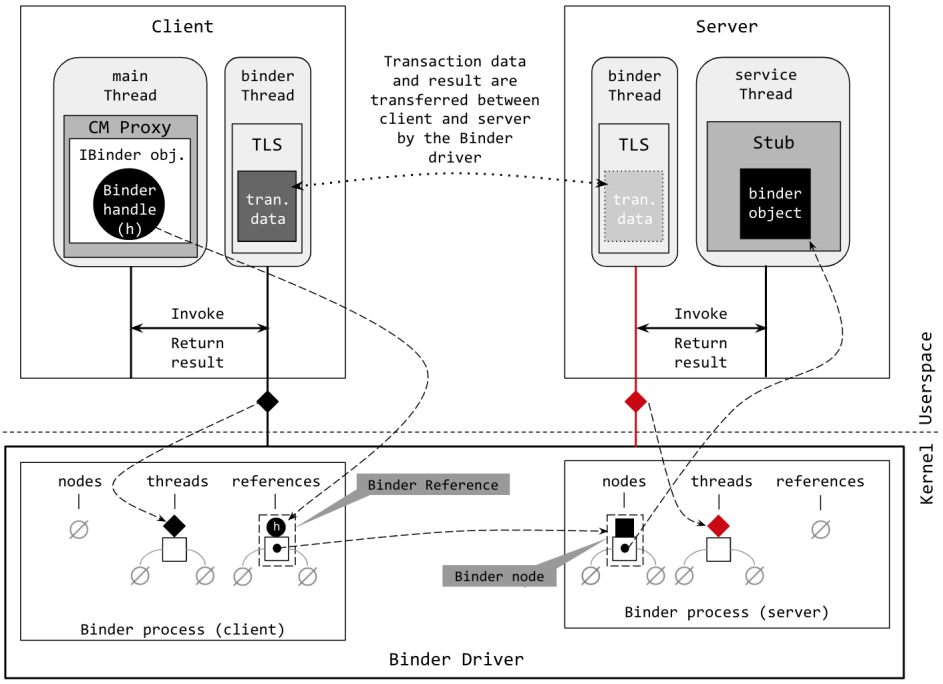


FIGURE 2.5: Binder Framework

The Binder is a term that is used to describe the overall IPC architecture, and it is a synonym for Binder framework. The core concept behind the Binder is inspired from OpenBinder project by Be Inc and later Palm Inc. In fact, portions of OpenBinder project was used in initial bringsup of Android [37]. However, the Binder framework, as implemented in Android, is completely specific to Android and has been adapted, extended, and integrated deeply into the platform. For example, Binder IPC is the primary, IPC mechanism used between apps and system processes (such as *system_server* and *servicemanager*) and between the apps themselves [32]. Moreover, Android 8 (Oreo) extends the Binder framework by introducing Binderized HALs which support Binder IPC between Android framework and HALs [25][39].

As shown in Figure 2.5, the Binder framework consists of many components that span over the kernel and the user space. In this section, we present those components and elaborate on their roles. Throughout the discussion, we use client and server to denote two processes, where the client wants to invoke an operation from a service offered by the server. The server, in turn, handles the request and returns a result, if any, to the client.

Binder Driver

The Binder driver is the most pivotal component in the Binder framework. It serves as a broker between any two communicating processes facilitating the client-server communication model. All processes willing to communicate over Binder IPC have to register IPC threads with the driver. Thus, the servers register worker IPC threads which

block waiting for requests from clients to serve. The clients, on the other hand, register IPC threads and use them afterward to initiate IPC requests to servers.

Assume a client wants to invoke an operation from a server, the flow of communication goes as follow:

1. The client uses an IPC thread to issue an ioctl call to the driver. This system call accepts a pointer to a data structure (called binder_write_read) as a parameter. This data structure contains a reference to the transaction data stored in the address space of client's IPC thread. As shown in Listing 2.5, the transaction data contains a handle to the target service (line 3), code of the target method to be invoked (line 7), and a data buffer with control information (lines 11-19). The data buffer holds method's parameters wrapped in a special object called a parcel [47].

2. The driver handles the ioctl call coming from client's IPC thread by saving client's thread information (to be used later on for returning the result) and copying transaction data to the kernel space (using *copy_from_user* system call). To associate the IPC request with the client, the driver injects client's identity (UID and PID) into the transaction data (lines 9-10). Using the handle of the remote service (line 3), the driver identifies the target server, populates the transaction data with specific information about the target service (lines 4 and 6), selects a thread from server's pool of available workers, and copies the transaction data (which, now, carries client's identity) to the address space of the selected thread (using *copy_to_user* system call). Finally, the driver wakes up (unblocks) the selected thread and passes it a reference to the transaction data stored in its address space.

3. The wakened thread reads transaction data from its address space, identifies the target object that need to be invoked (using the information in line 6), retrieves parameters and method information, and invokes the required method. If the IPC is one-way (this information is encoded in line 8), the IPC transaction ends at this point. However, if it is a two-way IPC, then the flow proceeds to the next, and last, step.

4. The server's thread issues an ioctl call to the driver and passes it a reference to the result in its address space. The driver copies the result from the address space of server's thread to the address space of client's thread that initiated the request (which is blocked waiting for a result). The driver then wakes up client's thread and passes it a reference to the result in its address space. At this point, server's thread blocks again waiting for further requests to serve, while client's thread is free to issue new IPC requests.

```
1  struct binder_transaction_data {
2          union {
3                  __u32   handle;
4                  binder_uintptr_t ptr;
5          } target;
6          binder_uintptr_t        cookie;
7          __u32           code;
8          __u32           flags;
9          pid_t           sender_pid;
10         uid_t           sender_euid;
11         binder_size_t   data_size;
12         binder_size_t   offsets_size;
13         union {
14                 struct {
15                         binder_uintptr_t        buffer;
16                         binder_uintptr_t        offsets;
17                 } ptr;
18                 __u8    buf[8];
19         } data;
20 };
```

Listing 2.5: Transaction Data

It worth noting that the transaction data stored in the address space of server's IPC thread is actually stored in a private memory block called Thread Local Storage (TLS). Each thread in Android has its TLS. The TLS guarantees that IPC threads, even those which belong to the same process, would not overwrite transactions data of each other [41].

Parcel

The parcel is a special container that is used to carry parameters between processes over Binder IPC. If a client wants to send the string "foo" to a server, the client has to create a parcel object and call *writeString("foo")* on it. Then, Android framework serializes parcel objects and attaches them into the transaction data. While handling the IPC request, the Binder driver copies the serialized parcel object from client's address space to the server's address space (as part of the transaction data). On the other side, the server reconstructs the parcel and calls *readString()* to get "foo". Parcel class supports all primitive types, such as int, string, float, and boolean, in addition to other complex classes, such as IBinder, Serializable, and Parcelable.

Binder Object

The high-level services (written in Java) that need to be accessible over the Binder framework have to extend a special java class, called *Binder.java*. Methods of this class enable services to identify callers, e.g., through *getCallingPid()* and *getCallingUid()* which

return the PID and the UID of the calling process, accordingly[11]. Additionally, this class provides a default implementation of a very important method, namely, *onTransact()*. This method is triggered whenever the service receives an IPC request. Services have to override this method to provide appropriate unmarshalling of transactions, more on this in *Subsection 2.3.1 Context Manager*

The Java classes of system services, such as LocationManagerService.java, PackageManagerService.java, and WifiService.java, extend the Binder.java class. As discussed earlier, these services, and more, are managed by the *system_server* process which, while booting up, creates an instance (Binder object) from each service class it manages and register it with the Context Manager (CM) to expose it to other apps. More on this later in the next section.

Each Binder object has a driver-level representation called Binder node (*binder_node*, see Listing 2.6). The Binder node contains information about the owning process (line 3) and references to the Binder object in server's address space (lines 5-6). When a server creates a Binder object and communicates it through the Binder driver (e.g., to register it with the CM), the driver creates a *binder_node* entry and stores it in the Binder process of the server.

```
1 struct binder_node {
2         {..}
3         struct binder_proc *proc;
4         {...}
5         binder_uintptr_t ptr;
6         binder_uintptr_t cookie;
7         {...}
8 };
```

Listing 2.6: Portions Of Binder Node Structure

Binder Handle

Binder handles are similar to file descriptors in UNIX systems. A handle is a key that references a low-level data structure. This data structure references a resource, namely, a Binder node. Binder handles are created and issued to user space processes (only upon their demand) by the Binder driver. On an abstract level, the client that possesses a Binder handle can use it to access an arbitrary operation from the remote Binder object referenced by the handle. The Binder driver resolves the handle and locates the server

[11]This is possible while the service is handling the IPC request and within the scope of the service. If these conditions are not met, the *getCallingPid() getCallingUid()* would return the PID and UID of the service process [41].

responsible for handling the request. The request is then forwarded to the server which invokes the required operation on the Binder object.

The Binder handle, as viewed from the user space, is merely a 32-bit integer value that is unique per process. However, Android framework builds an abstraction around this Binder handle and the result is an object of type IBinder. This object can be used to issue IPC requests to the remote service by calling its *transact()* method with the required parameters (which are encoded into a parcel object) and the code of the target method. When a transact method is called on an IBinder object, the Android framework extracts the Binder handle from the IBinder object and injects it, along with method's code and the parcel that holds the method's parameters, into the transaction data. Then, the client uses an IPC thread to issue an ioctl call to the driver which, in turn, transfer transaction data to server's side.

From the perspective of the Binder driver, each Binder handle is associated with a data structure that is called a Binder reference (*binder_ref*, see Listing 2.7), which contains information about the owner process (Line 3), the 32-bit handle value (Line 5), and the target Binder node (Line 4).

```
1 struct binder_ref {
2         {...}
3         struct binder_proc *proc;
4         struct binder_node *node;
5         uint32_t desc;
6         {...}
7 };
```

Listing 2.7: Portions of Binder Reference Structure

Binder Process

Each user space process involved in Binder IPC has a corresponding structure in the Binder driver called Binder process (*binder_proc*, see Listing 2.8). The Binder process references the user space process by its PID (line 6) and maintains three red-black binary trees for Binder references (line 5), Binder nodes (line 4), and IPC threads registered with it (line 3). The tree of Binder references contains all the Binder handles a process ever obtained, whereas the tree of Binder nodes references all Binder objects a process ever created. This entails that a process can be both a client and a server at the same time. In *Subsection 2.3.1 Context Manager*, we explore the relation between Binder references, nodes, and processes in more details.

```
1 struct binder_proc {
2         {...}
3         struct rb_root threads;
```

```
4         struct rb_root nodes;
5         struct rb_root refs_by_desc;
6         int pid;
7         {...}
8 };
```

Listing 2.8: Portions of Binder Process Structure

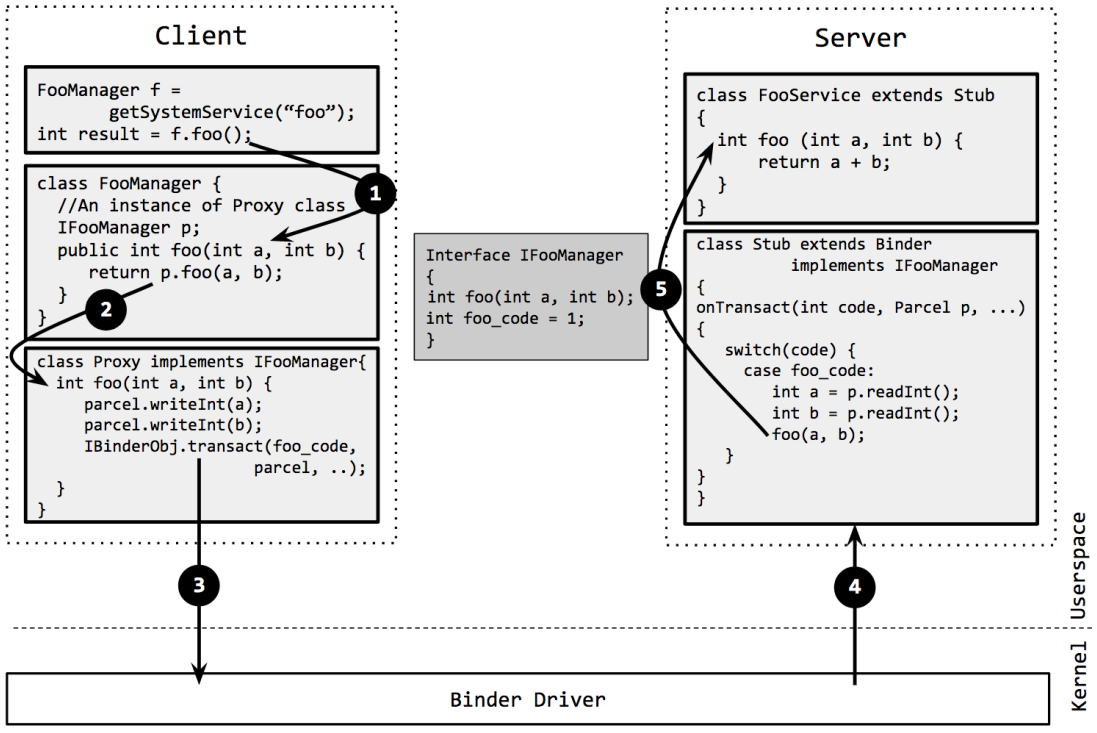


Figure 2.6: High-Level View on Proxy and Stub Classes

Proxies and Stubs

To abstract developers away from the low-level details on how Binder IPC transactions are handled, Android introduced Proxy and Stub classes. Everything starts with the remote service creating an interface that defines all of its remotely accessible methods. Both the Proxy that resides on client's side and the Stub that resides on server's side have to implement this interface.

The implementation of the Proxy marshals parameters of each method by encoding them into a parcel object. Then for each method, the proxy invokes *transact()* API (proxy classes extend IBinder class which provides a native implementation for this API). The Stub has to override the *onTransact()* method to unmarshal the data of incoming requests, by extracting parameters from the parcel, and then invoke the actual methods that implement the business logic of the service.

To understand how Proxies and Stubs are used, we consider a high-level scenario (depicted in Figure 2.6) in which an app uses the *getSystemService()* API to retrieve

a hypothetical manager (e.g., FooManager) that establishes an IPC connection with a hypothetical remote system service (e.g., FooService which exposes a single method). The manager encloses a proxy inside it. The proxy encloses an IBinder object which carries the handle value of the remote system service and offer the *transact()* method.

When the *foo()* method is invoked from the manager, the call is passed to the proxy which creates a parcel object and inject method's parameters in it. Then, the proxy calls the *transact()* method with method's code and data parcel as the first two parameters. On server's side, the request is received in *onTransact()*. Based based on method's code, the method extracts parameters and invoke the service which implements the required business logic.

It worth noting that if the operation is two-way (requires a result), then the *transact()* method blocks waiting for the result preventing the IPC thread from issuing other IPC requests. Moreover, the method's code passed in *transact()* (e.g., foo_code) is number which must be in sync between client and server. This number should be unique for each method in the same interface.

2.3.1 Context Manager

The primary prerequisite for processes to reach system services over the Binder framework is to possess Binder handles to them. For security and performance reasons, processes are granted handles to remote system services only upon their request. One approach to enable this design is to create a central unit that acts as a bookkeeper of system services. Processes on the other hand, must invoke this central unit and ask explicitly to be granted Binder handles to the desired system services. In Android, this central unit is the Context Manager (CM) or alternatively called the Service Manager.

The CM is a native daemon process that starts very early in the boot process of Android. While CM is booting up, it starts a loop with a handler function that listens on incoming IPC requests from remote processes through the Binder driver. The handler function supports a small set of commands, such as adding a service, getting a handle of a service, listing all registered services, etc.

Before executing any command, the CM checks the authorization level associated with the caller using SELinux and based on UIDs of callers. Depending on the result of the check, the CM will either reject or allow the execution of the command. For example, system processes (including system_server, drmserver, surfaceflinger, etc.) can add system services, whereas apps are explicitly prohibited from adding services; however, they can get handles of registered services. Additionally, isolated processes (which run

using a range of special UIDs [21]) are prevented from retrieving Binder handles.

Accessing The Context Manager

Although handles of system services are obtained through the CM, the handle of the CM service must be priorly known to all processes. While the CM is booting up, it instructs the Binder driver, over an ioctl call, to register itself as the only CM in the system. The Binder driver then creates a special Binder node, called binder_context_mgr_node, which holds a reference to the CM. Processes that need to invoke a function from the CM must set the target handle in the transaction data to *0*. When an IPC transaction with a handle of *0* reaches the Binder driver, the driver uses the special Binder node of the CM to retrieve a reference to the CM. Thus, the Binder driver passes the transaction to the CM, which in turn handles the request. This is especially important for processes when it comes to service registration and discovery, as processes can communicate with the CM directly using its universally known handle value.

In a more detailed level, Binder processes start with an empty tree of Binder references. In other words, a process starts with zero Binder handles in its possession. When a client process makes its first call to the CM, e.g. to add or get a service, the Binder driver queries the tree of Binder references for an entry with a handle of *0*. Since the CM has not been used before, no entry with a handle of *0* will be found. Therefore, the Binder driver creates a new Binder reference that points at the Binder node of the CM, sets its handle value to *0*, and inserts it into the tree. This happens only once during the life time of the caller process.

Registering System Services

As discussed in *Subsection 2.1.1 System Services*, system services are run and managed by a handful of middleware processes. When those processes start, they register their services in the CM. This serves as an announcement of their availability to be invoked by apps. One of the most important processes that does so is the *system_server* which is the focus of the following discussion.

To understand how services are registered in the CM, we need to distinguish between two data structures. The first one is the already known tree of Binder references that is stored inside the Binder process in the driver. The second data structure is a linked list that holds high-level information about services and stored in the user space of the CM process. Each entry in the service list of the CM contains a handle value, a name, and other control fields (see Figure 2.7).

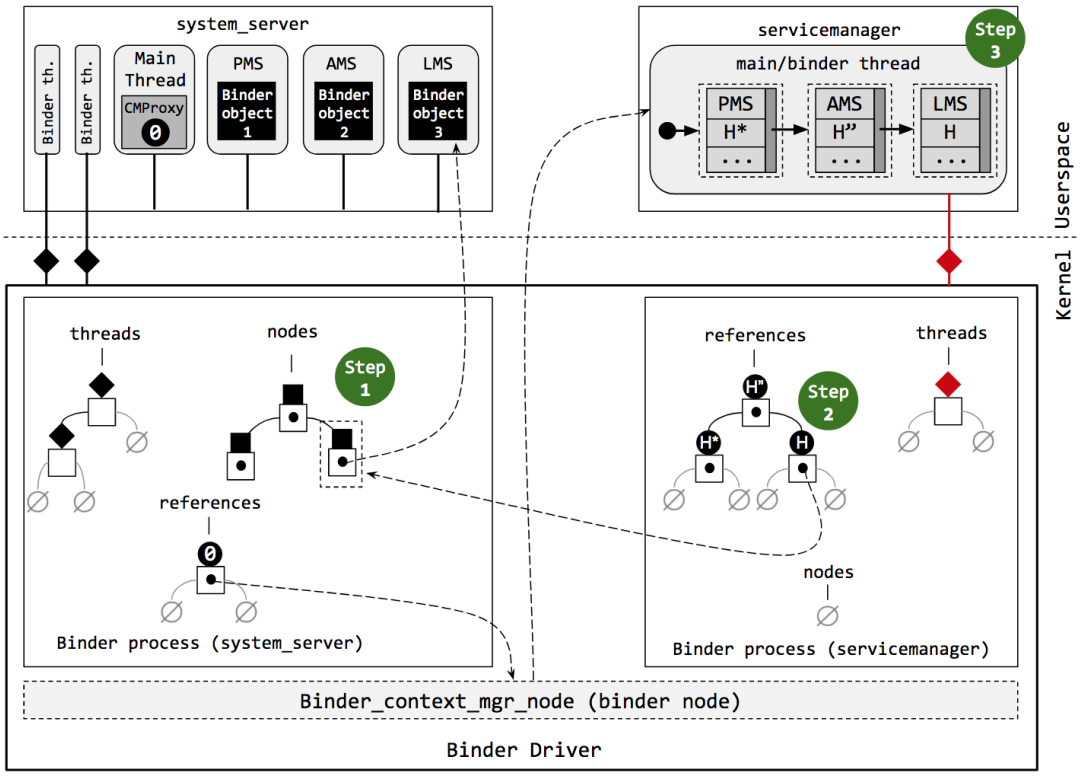


FIGURE 2.7: Registering System Services

When the *system_server* is ready, it instantiates an object (Binder object) for each service it manages. Then, for each Binder object, the *system_server* initiates an IPC to the CM. The transaction data will contain a Binder handle of value *0* (indicating that the CM is the target process of this transaction). Additionally, it will include a memory pointer to the Binder object, the name of the system service (and it should be unique over all services registered in the CM), the command code for adding services in the CM, among other data.

As shown in Figure 2.7, when transaction reaches the Binder driver, a new Binder node for the Binder object is created. The driver then updates the Binder node to reference the Binder object in the user space. The Binder node then gets inserted into the tree of Binder nodes that is specific to Binder process of the *system_server* (step ①). The CM must possess a Binder reference to the newly created Binder node. Thus, a new Binder reference is created with a handle "H" (step ②). Since the Binder driver acts merely as an intermediary and operations that require a high-level semantic should not be implemented in it, the high-level information such as the name of the service must be stored in the service list of the CM. Therefore, the Binder driver locates the CM and passes the transaction, including the handle value "H", to it. If the service has not been registered before, the CM creates a new entry in the list of services (step ③). The entry must include the handle value "H" and the name of the service. We observe that handle

values in the service list and the tree of Binder references of the Binder process of the CM must be in sync to reference the right Binder node. Additionally, handle values of system services must start from 1 as the 0-handle is reserved to the CM.

It worth noting that Android defines SELinux policies to control who can register services into the CM and what services are allowed to be registered. Thus, only a few processes are allowed to execute the command used for adding services in the CM. For example, *system_server* is the only process that can add the location service (and a bunch of other services), whereas *surfaceflinger* is the only process allowed to add *surfaceflinger* service. This security enforcement is essential, as untrusted processes cannot register malicious services and trick other processes into using their services as if they were the actual system services.

Retrieving Binder Handles

Clients that want to have access to system services have to go through the CM to retrieve Binder handles for those services. App developers can use the high-level API *getSystemService()* and pass it a name for the desired service. This API will take care of invoking the CM with the correct transaction data.

When the IPC request arrives the Binder driver, it forwards the request to the CM, which, in turn, searches its list of services for an entry with the desired name. If an entry is found, the CM returns the Binder handle (H) that is associated with the found entry to the driver. The driver searches the tree of Binder references of the CM for an entry with a Binder handle of value (H). When the entry is found, the driver duplicates it, assigns it a new handle value (the new handle value is calculated by adding one to the last issued handle), and inserts it into the tree of Binder references of the Binder process that belongs to the client. Finally, the driver returns the handle to the client.

Accessing Remote System Services

A client can access a system service if it possesses a valid Binder handle for it. The mechanics on how the driver handles service invocation is identical to the flows discussed earlier between clients/*system_server* and the CM. However, we think it is important to stress the point that system services can retrieve the UID and the PID of the calling process through *getCallingUid()* and *getCallingPid()* static methods of the Binder class, accordingly. Services use this information to check the permissions associated with each caller.

2.4 Problem Statement

In Android, processes and components of the same app share the same resources granted to the sandbox they belong to. This model possesses a risk as one buggy or vulnerable process/component could end up leaking sensitive information about the end user (e.g., leaking location, messages, and contacts) or performing unauthorized operations on her behalf (e.g., dialing phones, sending SMS, and altering files). To understand the extent of this problem, we consider the problem of 3^{rd}-party libraries.

Nowadays, developers tend to rely heavily on 3^{rd}-party libraries for app monetization, analytics, gaming, or reducing programming effort. Such libraries would run with the same privileges as of any other component in the sandbox. Given that such libraries come from arbitrary sources, there is a high probability that some libraries would misuse the power given to them, which in fact happens so frequently. Researchers [49][57] have detected some ad. libraries that access more resources than they publicly announce to developers and end users. Although several solutions have been proposed to address this problem, we believe there is still room for improvement.

As Android seeks to find a balance between security and usability for Android's permission system, they have introduced the concept of groups. Specifically, each dangerous permission must belong to a group. In turn, developers define the permissions they need from those groups. The system would grant all defined permissions of a group to an app if, at least, one member permission is explicitly granted by the user. Apparently, this enhances usability because users would not bother to know the details of each permission, and would not be confronted with security decisions now and then. On the downside, this model aggravates the problem discussed above, causing apps to live in overprivileged sandboxes (without user's consent for most of it) which make them attractive targets for exploitations by malicious 3^{rd}-party libraries.

Another shortcoming of the current security model of Android is its inability to track transitive invocations [30]. For example, when an app (A) invokes an interface (exposed unintentionally or deliberately) by another app (B) which, in turn, invokes an operation from a system service, the service will handle the request with the authority of app (B). This disregards the fact the app (A) might not have the required privilege to execute the operation itself. Malicious apps use this limitation to exploit unprotected (or poorly protected) interfaces offered by privileged apps (called confused deputy). Users of apps rely on app developers to protect their interfaces from being misused in such a way. In turn, app developers define permissions and require other apps to hold those permissions to be able to call their interfaces. This approach requires a high sense of security and

well understanding of Android's permissions system (which might not exist on some app developers).

From what we have presented in this section, we can easily come to the conclusion that the current security model of Android does not apply the principle of least privilege among components of the same app, which constitutes that a component/process should be given the least privilege required to achieve its functionality. In this work, we provide a new approach for limiting privileges of 3^{rd}-party apps on system services. We also propose a new mechanism that enables app developers to easily protect their interfaces that are deliberately exposed to other apps. Moreover, this work is moving towards reducing the effect of ambient authority that causes several problems in Android, such as the confused deputy.

Chapter 3

Related Work

In this section, we present a quick overview on object capabilities which serves as a preamble for introducing a hybrid system, called Capsicum, that leverages DAC, MAC, and capabilities to establish more confined sandboxes.

3.1 Object-Capabilities

A capability is a token that references an object and defines access rights on it [40], as shown in Figure 3.1. A subject that possesses a capability is qualified to access the object referenced by the capability. The access rights, associated with the capability, dictate what operations can the subject perform on the object. For example, a subject that has a capability to a file might be able to read it but not write it.

Each subject, e.g., user, process, or procedure, in a capability system is associated with a list of capabilities. This list contains all the capabilities ever issued to the subject. The system must protect the capability lists from being tampered with by any entity in the system. This is necessary to prevent subjects from escalating their privileges, e.g., by modifying the access rights or change the reference of the capability to access another object. Only the OS [48][45], or the hardware [51], can modify entries of capability lists.

To use a capability, the subject has to pass the index (handle) of the capability in an operation, e.g., WRITE(cap_handle, data). The system then retrieves the capability referenced by the handle cap_handle and decides, using the capability's access rights, whether the operation WRITE is allowed on the object or not. We can easily observe that the identity of the caller does not contribute to the decision. Thereby, capability systems eliminate the effect of ambient authority.

Processes can obtain capabilities through controlled channels with respect to the principle of least privilege. For example, a process can receive a capability after calling an OS routine. A capability can also be received by another process in the system (delegation). The system allow more operations on capabilities such as deletion (revocation), downgrading, and upgrading the access rights.

Finally, it worth noting that most modern operating systems provide implementations that resemble capabilities. For example, file descriptors in UNIX systems are considered capabilities. Each process has its own file descriptors list that is maintained by the OS. However, none of those systems is designed to support operations on capabilities, such as delegation, revocation, and access rights enforcement.

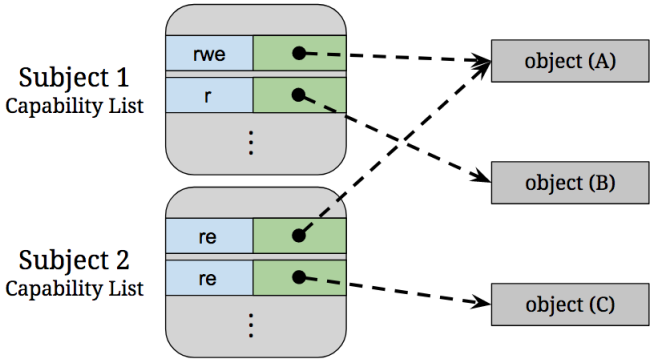


FIGURE 3.1: Capabilities and Objects

3.2 Capsicum: Capabilities in UNIX Systems

According to the authors of Capsicum project, Capsicum is a lightweight operating system capability and sandbox framework which extends, rather than replacing, UNIX APIs, providing new kernel primitives, namely, sandboxed capability mode and capabilities [50].

The motive behind this project is that UNIX systems are poor in applying the principle of least-privilege for running programs. The solution to this problem resides in using capabilities which define way more fine-grained access control than conventional UNIX systems. However, pure capability systems suffer from poor adoption, whereas UNIX systems are widely deployed. Therefore, the pragmatic solution is to introduce capabilities into UNIX systems as an extension forming a hybrid system that uses DAC, MAC, and capabilities for access control. This approach preserves existing UNIX APIs and performance, and presents application authors with an adoption path for capability-oriented design.

Capsicum extends UNIX file descriptors which possess some of the properties of object-capabilities as described in the literature: they are unforgeable tokens of authority, can be inherited by a child process, and passed between processes over IPC. Capsicum provides an API, namely cap_new, that takes a file descriptor and a mask of rights to create a capability. Each capability would encode roughly 60 possible mask rights. Each bit of this mask represents a permission, i.e., CAP_READ, CAP_SEEK, and CAP_IOCTL.

Capsicum enables two modes of operation: the normal mode, and the capability mode (which activated upon calling cap_enter). In the capability mode, processes are denied from accessing global namespaces (process IDs, sysctl, shm_open for named shared memory segments), in addition to other interfaces (i.e., /dev, ioctl, reboot and kldload). Access to system calls in the capability mode is restricted. For example, system calls that require access to global names spaces are prevented, while others (sysctl, openat, unlinkat, renameat) are constrained so that they can only operate on objects relative (but cannot contain ..) to the passed descriptor.

Upon invoking a system call on a capability, the kernel uses fget to retrieve the struct of a file descriptor which extended to hold access rights of the capability. This API serves as a single entry point for resolving file descriptors into references. Therefore, all operations on capabilities are guaranteed to be checked for access rights first. If the access rights allow the operation, then the execution will proceed. Otherwise, an error is reported and the API call fails.

As a straightforward example, we illustrate how Capsicum is used to sandbox the tcpdump application. The authors analyzed the program and found one system call, namely pcap_loop, that serves as a single entry point for processing packets from the network. The authors applied two changes; First, they modified pcap_loop to define the access rights which the passed file descriptors should have, i.e., writing operation requires CAP_WRITE. Second, they modified the tcpdump itself to confine the capability (which in fact is the file descriptor which pcap_loop will work on) and strip away unnecessary access rights from it before passing it to pcap_loop. When pcap_loop invokes a system call on the capability, the system uses fget API to resolve the target desired operation and required access rights. Then it retrieves the actual access rights the capability possess. Finally, it compares the acquired access rights, set (A), against the required access rights, set (B). If (A) is a subset of (B), then the access is allowed, and the execution proceeds, otherwise, it is prevented.

This approach has also been applied to other more complex applications, such as chromum, hdclient, and gzip.

Chapter 4

Design and Implementation

In this chapter, we introduce our design for supporting capabilities for system services with kernel-level enforcement. The design extends the current security model with new security features which are derived from capabilities as tokens of authority. Our goal is to mitigate the effect of ambient authority and effectively apply the principle of least privilege among app's components which require access to system services.

The security and efficiency of the proposed design are tied to the technologies presented in *Chapter 2 Technical Background and Problem Statement*. Specifically, we use the Binder framework as the building block for creating and communicating capabilities. We also rely on kernel's security guarantees to prevent forging capabilities. Additionally, we employ Android's permission model to reflect the dynamic high-level security decisions made by end users in order to encode the correct access rights into issued capabilities for system services. As a result, we fulfill our goal without significantly increasing the attack surface or causing a performance degrade.

We start the chapter by highlighting and justifying the key decisions which shaped our design. Then, we introduce the "big picture" that conveys the design and covers the prerequisite knowledge required to understand the following sections. Afterwards, we dive into the details related to the management of capabilities. Particularly, we discuss how capabilities of system services are created, delegated, revoked, and used for invocation. We end the chapter by talking about the different aspects of our implementation, such as the size of changes, scope, and limitations.

4.1 Design Decisions

As discussed in *Section 2.3 Binder Framework*, the Binder framework uses a distinguished architecture that produces unforgeable and transferable tokens, called Binder handles which are used as references to access Binder services. The unforgeability of Binder handles is guaranteed by the kernel, namely, the Binder driver, which we assume to be trusted and fortified against attacks. Although Binder handles are merely 32-bit integers (as viewed from the user space), processes can only communicate them over the Binder framework. Any attempt to transfer Binder handles over files, sockets, or any other IPC mechanism will be meaningless as the Binder driver would not be able to resolve the Binder handle when used for accessing Binder services. In other words, it is the kernel-level data structures maintained by the Binder driver that enables the usage of Binder handles as references.

All the aforementioned characteristics of Binder handles lead us to the realization that a Binder handle is, in fact, a capability that entails two states: The process can either access the Binder service or it cannot, based on either the process possesses a Binder handle to the Binder service or not, accordingly. This binary state is very coarse-grained in terms of access control. Therefore, for a Binder handle to be a fully fledged capability, it needs to encode the access rights of the owning process for the target Binder service.

Most Binder services are protected by high-level permissions. The owner of the device decides whether to grant an app, and transitively its processes, a specific permission or not. We sought that it would be for the best that we re-use those permissions to encode the access rights of the capabilities. The reason behind that is because we do not want to re-invent the wheel. Instead, we aim to instrument the current technologies to refine the access control mechanisms.

Following our discussion in *Section 2.3 Binder Framework*, system services are exposed using the same mechanism, i.e., they are registered with the CM which acts as a bookkeeper that processes must query to retrieve Binder handles to system services. On the other hand, bounded services are not registered with the CM. Due to time and effort limitations, we have decided to only support capabilities for system services and use Android's permissions to define the access rights of those capabilities. This decision does not roll out the possibility of supporting capabilities for bounded services (using user-defined permissions) because they all use the same technologies (e.g., Binder IPC and PMS). However, the design we are proposing is centric around the fact that system services are registered with the CM.

Moreover, other low-level resources which are accessed directly using system calls (such as file system and internet sockets) or using different IPC mechanisms than the

Binder IPC (such as local sockets and shared memory) are not considered for this work. Protecting low-level resources with capabilities requires low-level drastic changes to the system which need to be done carefully. Supporting capabilities for the high-level resources, such as system services, using the Binder framework can be considered as a complementary work to any effort for supporting low-level capabilities. Although it is not yet ported to Android, Capsicum is the best candidate for enforcing capabilities on low-level resources (see *Chapter 7 Future Work*).

In general, our design tends to borrow and reuse ideas from stock Android. For example, we slightly extended the Parcel class and used bounded services to delegate capabilities and revoke them afterward. In addition to reducing efforts, reusing and slightly extending components to achieve our goal comes with some benifits. Specifically, the proposed design keeps the attack surface almost intact and introduce unnoticeable performance overhead at some places. This performance overhead is evened, or even overcome, by performance gain at other places, later on this in *Section 6.1 Performance Analysis*.

Finally, throughout the following discussions **we use Binder handles and capabilities interchangeably**.

4.2 Big Picture

The goal of our design is to support capabilities for system services and provide a functional prototype. To effectively achieve this goal, we have to fulfill the following requirements:

- The system must be able to create capabilities for system services and delegate them to processes, only upon their request. However, the system must preserve the ability to upgrade and downgrade access rights of capabilities at runtime (this is necessary to keep up with dynamic permissions already used in Android).
- The capabilities which are delegated by the system[1] must encode the most up-to-date permissions of the recipient processes at any time. This is fundemental to prevent privilege escalation through capabilities.
- A process can have multiple capabilities that uniquely map to different system services. The access rights of one capability must encode the high-level permissions

[1] A capability can be delegated by the system and the processes. For the second case, the process that receives the delegated capability usually does not hold the permissions encoded in the capability. Otherwise, it can request its own capability from the system.

which are related to the system service referenced by this capability. For example, only the permissions that are related to the WifiService are used to encode the access rights of the capability issued for the wifi system service.

- A process must be able to use the capabilities it possesses as references to system services in order to access their functionalities. A system service, in turn, must be able to decode the access rights of the capability associated with the incoming request to decide whether to allow or reject the request. The implementation of this access control is solely based on the access right of the capability.
- Given capability's characteristics presented in *Section 3.1 Object Capabilities*, processes must be able to delegate capabilities to other processes to grant them a restricted access to system services. Moreover, the design must enable processes to revoke the capabilities delegated directly by them.
- As a good-to-have requirement, the design is preferred to provide the means for applying policies that control who can delegate, whether a delegated capability can be delegated again, and how the system should handle revocation of a delegated capability that has been re-delegated to other processes.

Before presenting the abstract communication flows of our design, we present four components that appear frequently throughout the discussion. Those components are:

Applications

Apps are active components that initiate all communication flows in our design. Processes of apps should explicitly ask the system to grant them capabilities for remote system services they wish to access. If a process has no permission to a specific system service and still asks for a capability for it, the system will issue it a capability with zero access rights. This means the process can only invoke the public[2] methods of the remote system service. Invoking a protected method in this case will raise a security exception. The moment an app receives a capability with non-zero access rights, it can use it to execute the protected methods of corresponding remote system service, or delegate it to another process with less or the same access rights of the capability.

System Server

Although system services are distributed among several processes (see *Subsection 2.1.1 System Services*), the *system_server* hosts most of these services and, therefore, we consider it for all communication flows. In our design, each system service must implement a reference monitor that merely relies on the access rights of the capabilities associated with the incoming requests. This is similar to the conventional approach for permission

[2]Public methods are normally security and privacy insensitive.

enforcement (discussed in *Section 2.2.2 Android's Permission System*), where system services implement a per-method reference monitor. However, in the conventional permission system, system services use the UID and PID of the incoming request and consult another system service, namely, the PMS, to decide whether to allow the request or not. On the other hand, capabilities enable in-place access control without relying on any knowledge stored anywhere other than the access rights associated with the capability of the incoming request. The check itself is cheap and performed using bitwise operations on the access rights. Throughout the following discussion, we assume that all system services are already registered into the CM.

Context Manager

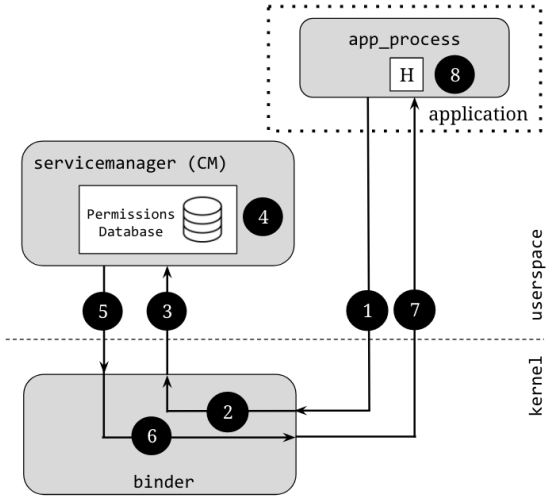
In stock Android, the Binder driver requests[3] a handle value from the CM for a specific service name and for a specific process. In turn, the CM returns the handle value of that service. We extend this flow by making the CM return the handle value and the access rights which the process's sandbox has on the target system service referenced by the handle. We extend the functionalities of the CM to supply it with the high-level permissions necessary to compute the access rights of the capabilities issued to processes. Our design makes sure that changes on the high-level permissions is reflected directly in the CM. This guarantees that the computed access rights are up-to-date and no privilege-escalation or downgrading would unintentionally happen. For the sake of brevity in this section, we assume that all permission information granted to all sandboxes is already reported to the CM which is now able to compute the correct access rights for issued capabilities.

Binder Driver

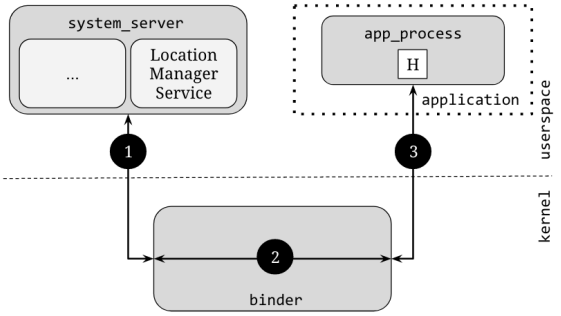
Each IPC transaction in the Binder framework goes through the Binder driver, see *Section 2.3 Binder Framework*. This makes the Binder driver the perfect component to introduce our changes for capability management (more on this later in this section), such as delegation, revocation, and assigning access rights to capabilities. However, the changes need to be lightweight to not affect the overall performance of the Binder framework. The Binder driver is keeps track of all capabilities and the associated access rights issued to each process. This enables the delegation and revocation of capabilities.

We should mention that we have two modes of operation: capability and normal modes. The mode of operation is applied globally based on a boolean value, namely, isCapMode. For example, in the capability mode, all processes that acquire service handles to the LMS would be subject to access control using capabilities as opposed to permissions access control in normal mode.

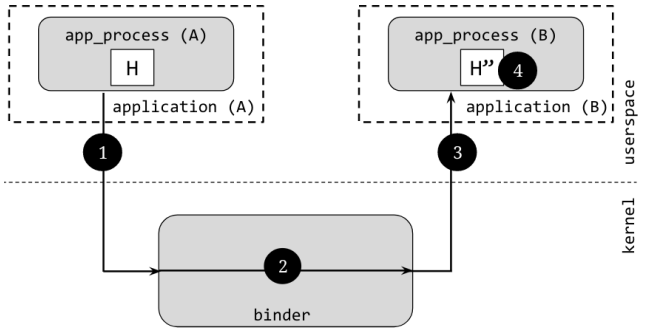
[3]This request is made on behalf of the calling process.



(A) Acquiring a capability for a system service



(B) Invoking the system service using the acquired capability



(C) Capability Delegation

FIGURE 4.1: Abstract Flows For Acquiring, Invoking, and Delegating Capabilities

Next, we present the abstract communication flows for acquiring, delegating, revoking, and invoking capabilities. In this section, we refrain from discussing some corner cases that require special handling by our design. Instead, we discuss those special cases in the in the next section.

Acquiring Capability

The Figure 4.1a, depicts the abstract communication flow used to acquire a capability for a system service. The flow starts by a process invoking the *getSystemService()* API from the application framework (step ①), which initiates an IPC to the CM process. In stock Android, this API returns a Binder handle to the system service (see *Section 2.3.1 Context Manager*). In our design, this API still returns a Binder handle but with more fine-grained access rights associated with it, forming a fully fledged capability. The

Binder driver, as an intermediary for all IPC calls, intercepts the call, injects callers identity (PID and UID), and forwards the request to the CM process (steps ② and ③). The CM uses its up-to-date database of permissions to compute the access rights of the process on the desired system service which is referenced by its name (step ⑤) and returns the handle of the service along with the computed access rights to the kernel (step ⑥). Finally, the kernel creates a new Binder handle (internally, it is a Binder reference), injects the access rights returned from the CM inside it (step ⑦), and returns the Binder handle to the calling process (step ⑦). Eventually, the app process will have a Binder handle (step ⑧) that references a remote system service and encodes access rights in it.

Capability Invocation

As shown in Figure 4.1b, app's process can now use the handle it received from the CM to access the target system service. For simplicity, we assume the handle (or capability) is for the LMS. The flow of invoking a method from the LMS starts with the process initiating an IPC to the *system_server* process (step ①). However, the request passes through the Binder driver which injects callers access rights, and identity (PID and UID), into the request before forwarding it to the *system_server* (steps ② and ③). The *system_server* dispatches the request to the LMS. The LMS then uses the newly introduced API *getCallingCapability()* to retrieve the bit mask that encodes caller's access rights. Then, it performs an in-place check (step ④) to decide whether the calling process has the required permission to execute the target method or not. Finally, the request is either served and the result gets back to the caller, if any, or a security exception is raised indicating that the caller does not have the required permission.

Capability Delegation

Any process that possesses a capability with non-zero access rights can delegate it to other processes over the Binder framework. The Figure 4.1c shows how the process (A) delegates the capability (H) to the process (B). The outcome of this operation must be that process (B) has a capability (H") which references the same object referenced by the original capability with the same or less access rights of the original capability. Capabilities can only be transferred (delegated) over the Binder framework. Therefore, both processes, (A) and (B), have to establish a client-server Binder IPC channel. The process (B) is the server while process (A) is the client. Additionally, process (B) have to expose an API for process (A) so the capability can be sent through it. If everything is set, the flow of delegation goes as following: Process (A) invokes the interface exposed by process (B) and attaches the capability and the desired access rights (step ①). The Binder driver intercepts the transaction and check for validity of the delegation request. If the delegation is permissible, a new capability (H") is created for process (B) with the passed access rights (step ②). Then, the driver sends the new capability to process (B)

(step ③). Finally, process (B) receives the capability (H") and can use it to access the associated system service as described earlier with respect to the access rights associated with it.

Capability Revocation

When it comes to the revocation of capabilities, we have to consider two scenarios caused by two different events:

1. Revoking capabilities by the system: This scenario happens when the user revokes a permission group from an app using the the "Settings" system app. For this scenario, the revoked permissions are reported to the CM and the app is restarted, only if it was already running (this actually happens in stock Android). Since capabilities are bound to processes, killing a process will delete all the capabilities associated with it. Therefore, the restarted process has to request new capabilities for system services. At this point, it is guaranteed that any new issued capabilities will reflect the most up-to-date privileges granted to the requester process.

2. Revoking capabilities by apps: This scenario happens when a process (A) delegates a capability to another process (B) and then wants to revoke it. In this case, revocation is not implemented through deletion but as reducing of access rights. For example, process (A) can use "0" as the new access rights, dictating that process (B) no longer can access the protected methods of the target system service. It worth noting that this method can also be used to upgrade the access rights of the capabilities delegated by processes.

4.3 Management of Capabilities

In this section, we cover in details how capabilities are obtained and how access rights are computed and attached to it. Further, we discuss how to delegate, revoke, and invoke capabilities.

4.3.1 Capabilitys Access Rights

While presenting the "big picture" of our design, we assumed that the CM has the most up-to-date knowledge that is necessary to encode the high-level permissions into access rights for the capabilities issued to every process. In this subsection, we elaborate on how this is done and why the CM has been chosen for this task in the first place.

The process of creating capabilities requires some logic for encoding the high-level permissions into a limited-size variable which is attached to each capability. The entity responsible of the task of encoding access rights, needs to possess information about all the high-level permissions granted to each app. This seems very fitting task for the PMS which maintains a hash-map of all apps and their permissions. However, capabilities are issued per-process while the PMS stores permissions per-UID, and its logic for mapping PID to apps is not straightforward (has to go through the AMS). Furthermore, the PMS is not involved in the process of acquiring a capability, and consequently, the driver must initiates an IPC to the PMS which, in fact, this drastic changes because the driver is a passive component that only forwards requests and do not initiate IPC requests itself. Based on that, the PMS is no longer a candidate for the task of encoding access rights.

The next component to consider is the Binder driver which only deals with processes and does not have any information about high-level permissions associated with apps. Therefore, if Binder driver should handle the task of encoding access rights, we have to supply it with the high-level information and it should store that information in the kernel space. However, since the Binder driver is involved in all Binder IPC transactions, it should not contain any complex logic that could possibly cause unnecessary overhead on all Binder IPC transactions. Moreover, it is a bad practice to encode high-level semantic in kernel drivers. Therefore, the Binder driver cannot be considered for this task.

This leaves us with the CM, which has been chosen for this task for several reasons:

1. It has a notion of processes and apps at the same time. For example, the CM in stock Android prevents isolated processes that run with special UIDs.
2. Even though it does not have any information about the high-level permissions, it can be extended easily so this information is supplied to it.
3. It acts as a single entry point for processes that require handles to system services registered with it. The point of time when a Binder handle is returned should logically be the time when the access right is computed.

The aforementioned reasons, makes the CM the perfect component to compute the access rights of each process. Therefore, the next task would be to supply it with the knowledge of the high-level permissions assigned to each app.

We have extended the CM to support a new command (in addition to the commands used for registering and getting services). Similar to how services are registered and retrieved, the PMS invokes the new command of the CM whenever a new permission is

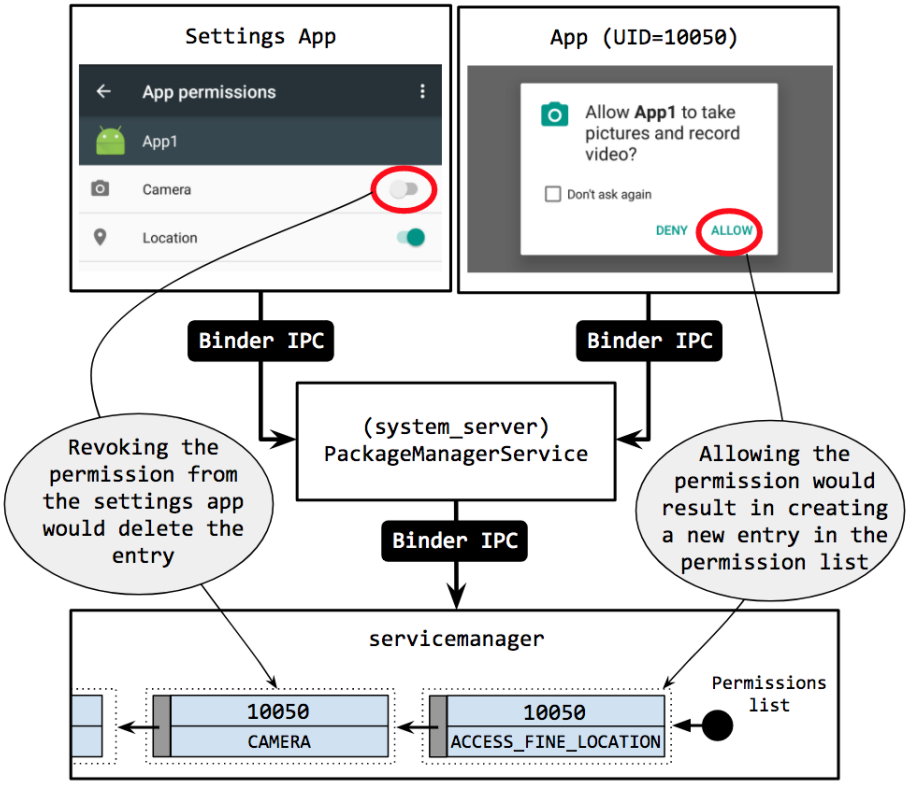


FIGURE 4.2: Reporting Permissions

granted or revoked. The parameters for this call are the UID of the app, the permission name, and a boolean value that indicates whether the action is granting or revoking.

The CM maintains a linked list that holds only granted permissions reported from the CM, without duplication. If a permission is granted then revoked, its corresponding entry in the list maintained by the CM is removed. Given that the PMS get notified when an app is deleted, it revokes all permissions associated with the app. Therefore, it is guaranteed that the CM will keep permission information for installed apps only.

The Figure 4.2, shows a high-level view on how permissions are reported from app and settings app to the CM.

Access Rights Encoding

When all permissions for each app is reported to the CM, then it becomes feasible for the CM to compute the access rights for each process on the target capability. Our design imposes a limitation on the size of the variable where the access rights will be encoded in, specifically, it is 32 bits. This means, only 32 permissions can be encoded per service where each bit represents a granted/not granted state of a high-level permission.

For the CM to be able to encode the access rights, it has to know what permissions are used in each service. Therefore, we ran through the LMS (and associated classes) and found that it uses 6 permissions to control access to its methods. We have done the same for another WifiService and found that is uses only three permissions (see Table 4.1). We made those groups available to the CM by statically writing them in its code. The order for which those permissions decides the encoded access rights. For example, assume an app is granted permission (1), (3), and (6). Then, the equivalent bitwise access right is 0xA200. The logic of encoding of the access rights in the CM must be in sync with the decoding logic on the service side. Otherwise, undesired side effects of privilege escalation and prevention will occur.

We should mention that permissions are stored with the service name as registered in the CM. This enables the CM to identify only the permission for a specific service.

Table 4.1: Permissions Required Per Service

Service	**Index**	**Permission**
LMS (location)	0	ACCESS_COARSE_LOCATION
	1	ACCESS_FINE_LOCATION
	2	ACCESS_LOCATION_EXTRA_COMMANDS
	3	CONTROL_LOCATION_UPDATES
	4	INSTALL_LOCATION_PROVIDER
	5	LOCATION_HARDWARE
WifiService (wifi)	0	ACCESS_WIFI_STATE
	1	CHANGE_WIFI_MULTICAST_STATE
	2	CHANGE_WIFI_STATE

4.3.2 Acquiring Capabilities

When the system is in the capability mode, processes use the *getSystemService()* API which invokes the *getService()* API from the ServiceManagerProxy to acquire a capability for the desired remote system service referenced by a name supplied to the API. The same API is used, when the system is in the normal mode, to acquire a Binder handle to a system service. In both cases, the returned result is the same, which is merely a handle to a system service. However, when the system is in the capability mode, a special kernel-level arrangements are done on the Binder handle to turn it into a capability. Those arrangements include extending the structure of the Binder handle to hold the access rights of the owning process, and introducing more computational logic on the extended Binder handle (i.e., for delegation, revocation, and invocation).

Figure 4.3 depicts the detailed flow for acquiring the capability for the LMS. The flow builds on the knowledge we have accumulated so far. Specifically:

1. The *system_server* process instantiates all of its services and registers those instances (called Binder objects) into the CM to make them accessible over the Binder framework. We assume all services, including the LMS, have already been registered.

2. The client process in this flow, namely the app, can access the CM directly. However, it cannot yet access the LMS because it does not possess a Binder handle to it.

3. A Binder handle is simply a Binder reference that points to the target Binder node. The Binder reference is stored in a special data structure on clients Binder process.

4. The CM collects the information needed to compute the access rights of the capabilities issued to each calling process. This information is always up-to-date to ensure the correctness of the computed access rights.

Compared to the process of acquiring a Binder handle to a remote system service (as discussed in *Subsection 2.3 Binder Framework*), only two changes are introduced by our design. Those changes are as follow:

1. When the *getSystemService()* request reaches the CM, it uses the UID of the caller to retrieve all permissions associated with the desired service name. Then, for each permission, it sets or clears the corresponding bit in the a 32-bit variable that would eventually encode all access rights. Finally, the CM returns the encoded access rights and the handle value of the desired service to the Binder driver.

2. The Binder driver searches the tree of Binder references, stored in the Binder process of the CM, and locates the Binder reference that has a handle equivalent to the handle value returned from the CM. The Binder driver then duplicates this Binder reference, injects the access rights returned from the CM into it, changes the handle value of the new Binder reference, and then inserts it inside the tree of Binder references that belong to apps Binder process. Finally, it returns the handle value to apps process.

App's process can now use the handle to invoke methods of the LMS. Next, we discuss how the Binder driver facilitates that.

4.3.3 Capability Invocation

In comparison to the flow discussed in *Subsection 2.3 Binder Framework*, accessing a remote system service in the capability mode has two significant differences. Those differences go as follow:

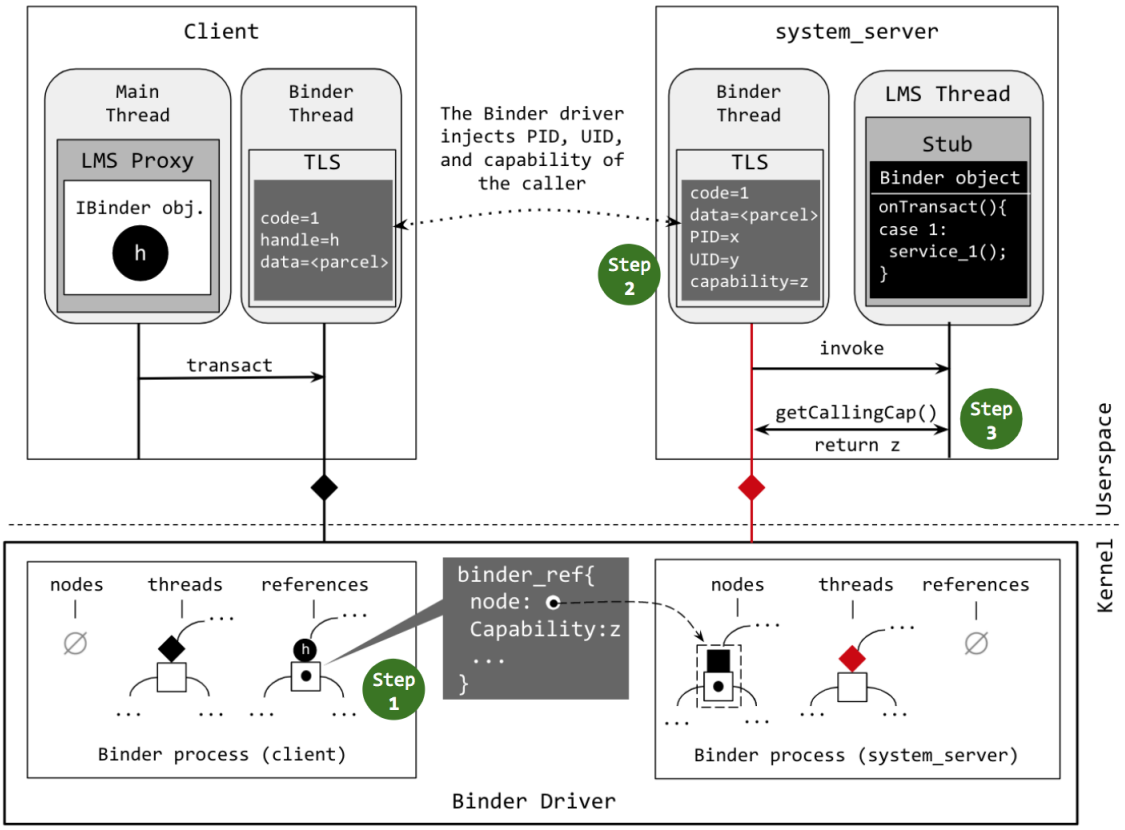


FIGURE 4.3: Requesting Capabilities

1. The driver intercepts the invocation call from the app's process, locates the Binder reference associated with the Binder handle of the request, retrieves the access rights and the reference of the target service from the Binder reference (step ①), locates an available IPC thread for the target service, injects caller's identity into the transaction data (including the access rights), and finally copies transaction data into the TLS of the server's IPC thread (step ②). The transaction data then becomes available to the IPC thread handling the request on the server side.

2. When the request reaches the *system_server*, it dispatches the transaction to the LMS. While handling the request, the LMS can call the *Binder.getCallingCapability()* API to retrieve the access rights of the calling process from the TLS of the IPC thread that carried the transaction (step ③). The LMS performs bit-wise operations on the access rights of the calling process to see if the caller can execute the target method.

As opposed to the conventional reference monitoring on the server side which requires consulting the PMS[4], only one cheap arithmetic operation is needed in the proposed

[4]the *system_server* hosts both the LMS and PMS which means no IPC is required for this case. However, other services, such as CameraService, could live on another process causing an IPC request to issued for each permission check.

design to decide on allowing or rejecting the request (see Listing 4.1).

```
1  int ACCESS_COARSE_LOCATION_MASK           = 0x0001;
2  int ACCESS_FINE_LOCATION_MASK             = 0x0002;
3  int ACCESS_LOCATION_EXTRA_COMMANDS_MASK   = 0x0004;
4  int CONTROL_LOCATION_UPDATES_MASK         = 0x0008;
5  int INSTALL_LOCATION_PROVIDER_MASK        = 0x0010;
6  int LOCATION_HARDWARE_MASK                = 0x0020;
7
8  private int getAllowedResolutionLevel(int pid, int uid) {
9      int capability = Binder.getCallingCapability();
10     if ((capability & ACCESS_FINE_LOCATION_MASK) != 0) {
11         return RESOLUTION_LEVEL_FINE;
12     } else if ((capability & ACCESS_COARSE_LOCATION_MASK) != 0) {
13         return RESOLUTION_LEVEL_COARSE;
14     } else {
15         return RESOLUTION_LEVEL_NONE;
16     }
17 }
```

Listing 4.1: Reference Monitor On Capabilities

4.3.4 Capability Delegation

In stock Android, a process can transfer a Binder handle to another process over the Binder framework. It would be very beneficial if we can re-use the same flow of transferring Binder handles for capability delegation for the following reasons: First, we would benefit from the security model of the Binder framework that prevents malicious processes from interfering in the IPC, e.g., by changing the target of the delegation and access rights. Second, the performance of the delegation process would be nearly as efficient as any Binder IPC transaction.

The application framework enables the transmission of Binder handles by providing the *writeStrongBinder()*[5] API from Parcel class which takes an object of type IBinder. This API attaches the IBinder object to the parcel that carries other transaction data which need to be sent to the target process. Figure 4.4 depicts the flow which takes place when a process sends a Binder handle to another process. The flow goes as follow:

1. Both sender and receiver of the Binder handle must establish a Binder IPC channel. The sender (client) holds a proxy for the remote Binder object that encapsulates the service offered by the receiver (server).

[5]In addition to transferring Binder handles, app developers can use this API to transfer Binder objects between processes. In both cases, the API does not write the object itself. However, it writes the value of the Binder handle or the token that references the Binder object in the user space of the owner process. The receiver process will get a Binder handle regardless of what the sender has originally sent.

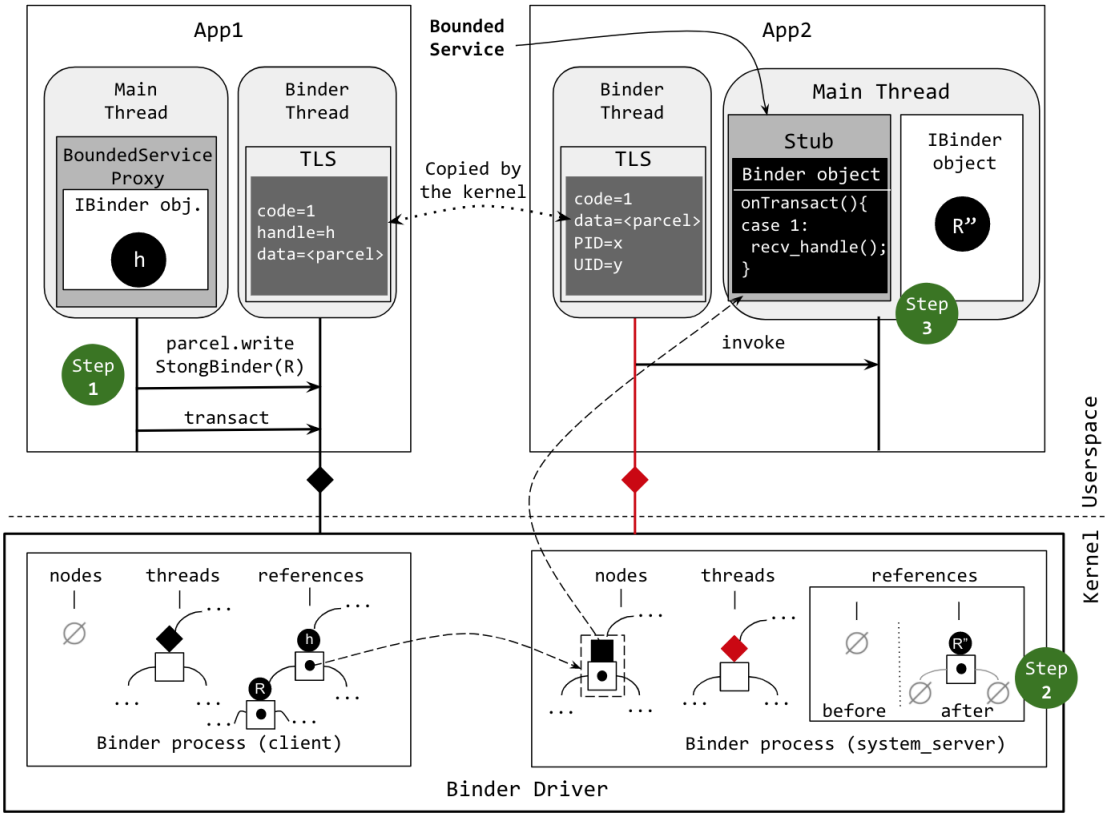


FIGURE 4.4: Transferring Binder Handles Between Apps

2. The client attaches the Binder handle to the parcel that needs to be sent to the receiver process using *writeStrongBinder()*, and then invokes an operation from the remote Binder object using its local proxy (step ①). This operation expects to receive a Binder handle.

3. The request reaches the Binder driver which recognizes that transaction data contains a Binder handle which the receiver process should have. Thus, the driver searches the tree of Binder references of the sender Binder process. Then, it locates the Binder reference (R) that has a handle value that is equal to the value of the transmitted Binder handle. Then, the driver duplicates (R) into (R”), changes the handle of the duplicated entry, and inserts (R”) into the tree of Binder references of the receiver Binder process (step ②).

4. The Binder driver then forwards the request to the receiver process which in turn extracts the Binder handle from the parcel (step ③).

Enabling app developers to use this mechanism for delegating capabilities imposes three challenges:

First, the application framework builds a manager around the Binder handle that references the system service. The framework hides the methods used to extract Binder

handles from the managers. However, we need such functionalities from application framework to enable delegation. Thus, we used Java reflection to extract the Binder handle from a manager at sender's side. On receiver's side, we use the reverse operation using reflection to build the manager. We preferred not to expose the hidden methods from application framework as that would be a bad practice and might open the door for security issues. In Listing 4.2, we present the code used in extract Binder handles from the WifiManager and build the managers again.

Second, we need to create an API that is similar to *writeStrongBinder()* but accepts an extra parameter that represents the access rights to be delegated. The new API is called *writeDelegatedStrongBinder()*. Note that cannot simply attach the access rights as a parameter in the parcel because then it would be serialized in a data buffer because the Binder driver then, has to look inside the buffer to extract the access rights, which would be inefficient. Instead, we attach the access rights to the transaction data as a separate attribute.

Third, we need to introduce the delegation logic in the Binder driver, in which the driver does not only create a Binder reference in the receivers Binder process, but also attaches the new access rights after checking them for validity into it. The validity check makes sure that the set of delegated access rights is a subset of the original access rights acquired by the sender.

```
1  public IBinder extractBinderHandle(WifiManager wifiManager) {
2      IBinder binder = null;
3      try {
4          Class wifiManagerClass = wifiManager.getClass();
5          Field mServicefield = wifiManagerClass.getDeclaredField("mService");
6          mServicefield.setAccessible(true);
7          Object proxy = mServicefield.get(wifiManager);
8          Field mRemotefield = proxy.getClass().getDeclaredField("mRemote");
9          mRemotefield.setAccessible(true);
10         binder = (IBinder) mRemotefield.get(proxy);
11     } catch (Exception e) {
12         e.printStackTrace();
13     }
14     return binder;
15 }
16
17 public WifiManager constructWifiManager(IBinder binder) {
18     WifiManager wm = null;
19     try {
20         Class iwmStub =  Class.forName("android.net.wifi.IWifiManager$Stub");
21         Method[] aMethods = iwmStub.getDeclaredMethods();
22         for (Method method : aMethods) {
23             if ("asInterface".equals(method.getName())) {
24                 method.setAccessible(true);
25                 Object iwm = method.invoke(null, binder);
```

```
26                  Class wmClass =  Class.forName("android.net.wifi.WifiManager");
27                  for (Constructor constructor :  wmClass.getDeclaredConstructors()) {
28                      wm = (wm) constructor.newInstance(
29                          RemoteService.this.getBaseContext(),
30                          iwm, null);
31                      break;
32                  }
33              }
34          }
35      }  catch (Exception e) {
36          e.printStackTrace();
37      }
38      return wm;
39  }
```

LISTING 4.2: Reflection To Extract Binder Handle From Manager And Constructing The Manager Again

As a prerequisite for the delegation, the process that is willing to delegate must have a notion of the access rights associated with the Binder capability it possesses. This can be done by calling a *getAccessRights()* API which returns a bit-mask that encodes the access rights.

This implementation can be extended to keep track of each delegated capability (by extending the Binder reference with the parent/children). This is especially helpful for an advanced logic of revocation, i.e., revoking a capability results in revoking all capabilities that have been instantiated from it. Additionally, Binder capabilities can be extended to carry a flag that prevents delegation.

The same channel used for delegation can be used again to downgrade or upgrade the access rights of the delegated capability. As we will discuss next, we use the downgrading of access rights as a special form of revocation.

As a final note, we can observe that the CM is not involved in the delegation flow and it is up to the delegator process to decide what access rights can be transferred to the receiver process. In turn, it is the responsibility of the Binder driver to correctly compute the access rights of the receiver of the delegation based on the access rights of the sender.

4.3.5 Revocation of Capabilities

There are two techniques for revocation triggered by two different events in the system. The first technique is through actual capability deletion. The second technique is through downgrading the access rights of the delegated capability to zero. Implementing

revocation through downgrading of access rights implies that a process would still have a reference to the remote system service. However, it cannot access its protected methods.

Revoking Capabilities via Settings App

As discussed in *Subsection 2.2.2 Android's Permission System*, starting from Android 6, Android allows users to revoke permissions by groups. When the user decides to revoke a permission group, the app is killed, gracefully. Before the app is killed, the system takes memory snapshots of the running processes of the app and then kills all of them. The system then instructs the PMS to revoke all permissions of the group. The PMS would then report this change in permissions to the CM which would remove all revoked permissions from its list). If the app is opened again, the system forks a new process from zygote, loads apps code, and restores the corresponding memory snapshot into the memory.

Revoking/Downgrading Delegated Capabilities

Processes that delegate a capability to another process can change the access rights associated with the delegated capability based on an internal logic using the same channel used for delegation. If the access rights are set to zero, then we call that a revocation of the capability.

To change the access rights of a delegated capability, the process has to use the already established delegation channel. This means, the delegator process have to use the *writeDelegatedStrongBinder()* for the same Binder handle but with new access rights. This method works because each process can only have one capability for each system service at any point of time. The new access rights which is passed in the API will overwrite the access rights which the target process originally has, if any, on the corresponding system service. The Binder driver has to make sure that the new degraded access rights are a subset of the access rights that the delegator process originally has.

4.4 Implementation

In this section, we present the scope of our design and the limitations imposed by our implementation. Then, we provide a brief analysis of the amount and type of changes we introduced.

4.4.1 Scope and Limitations

The capabilities introduced by our design are specific to the system services that fulfill the following condition: they are registered in the CM and accessible over the Binder

framework using the *getSystemService()* API. Consequently, the design does not support capabilities for services that are not registered in the CM, such as user-defined services. Moreover, the *getSystemService()* API does not retrieve all services registered with the CM. For example, developers have to use the ContentResolver to access the content system service, and to record audio, developers use the RecordAudio class (which uses the *surfaceflinger* service).

Among all system services that fulfill the condition mentioned above, we decided to adapt the LMS and WifiService to use capabilities for access control. We have built two apps that serve as a prototype. The first app request all dangerous and normal permissions to both services and delegate access to both to the second app. The second app makes calls to those services which control the access using permissions. Due to time limitations, we could not adapt more services. However, we performed an analysis on all system services protected by permissions, and the results show that we can apply capabilities on 18 other system service which are protected by 60% of the permissions defined in the system. More on this in *Subsection 6.2 Coverage and Effectiveness*

Since we rely on Android's permission system for creating access rights on capabilities, we report all granted and revoked permission from PackageManagerService to the CM. We do not exclude any permission from being reported even though some normal permissions, like android.permission.INTERNET, are enforced by DAC *Subsection 2.2.2 Android's Permission System.*

On another point, we do not implement a global switch that controls what mode the system is in, i.e., capability mode or normal mode. This switch is currently distributed between kernel and user space components. Furthermore, we do not implement switching between modes at runtime. Instead, the system (kernel and AOSP) must be configured to use a specific mode before building it.

Regarding acquiring capabilities, the developer should request a capability, using *getSystemService()* API, after she is granted the permission(s) to the target service. This is because access rights of the capability is attached to Binder reference in the kernel only upon calling this API.

As for revoking delegated capabilities, we do not implement a way for actually deleting the delegated capability. Instead, we enable the app developer to reduce the access rights of the delegated capability to 0 based on apps logic.

We do not enforce a policy for delegation across apps. This magnifies some attack scenarios as we would discuss in the following chapter. Another less serious issue related to the fact that when a capability is delegated, the sender has to take care of extracting the Binder handle from the manager using reflection before sending the handle to the

receiver who, in turn, must reconstruct the manager out of the delegated capability, also using reflection. This requires a detailed knowledge of the class structure of the manage. We did not have time to implement a simpler way to extract Binder handles and reconstruct manager without using reflection.

During early stages of this work, we tried to build the Goldfish kernel of version 4.4 and use it with AOSP of Android 7. Although both builds succeed, we could not manage to run the default Android emulator of AOSP. We tried several configurations and versions with the same failing outcome. Eventually, we managed to run goldfish v.3.4 against AOSP of Android 6.1 and Android 7.1.2. We observed that no drastic changes are done on the Binder kernel module. Therefore, we believe our approach would work on any version of Goldfish. However, we have not tested that.

4.4.2 Changes on AOSP and Kernel

Before we started the implementation phase, we minutely explored Android's platform and kernel. This helped us afterward to reduce the amount of code used to come up with a working prototype. Since we touched several layers of Android's software stack, we had to write code in three programming languages. In total, we have written 756 lines on codes (LoCs). Table 4.2 shows the number of LoC per language and project. We calculated the number of LoCs manually after extracting the patch files from AOSP (using repo) and kernel (using git). As we excluded comments and lines of code used for logging, we believe our prototype will not work without any of these lines.

Table 4.2: LoCs Introduced By Our Design

Project	**Language**	**LoC**
AOSP	Java	310
	C++	156
	C	260
Goldfish Kernel	C	30

In general, changes in Java introduce new APIs to app developers used for acquiring a capability, attaching a capability in a parcel delegation, query about access rights of a capability (see Table 4.3). Other internal APIs are used for reporting permissions to the CM, extracting access rights from IPC threads, etc. C++ changes, in most cases, are meant to transfer requests made from Java to the kernel and back. One of the important changes are those made to the IPCThreadState.cpp which provide the necessary API for the system services to call, over JNI, to retrieve the access rights of the calling process from the IPC thread.

C changes in AOSP are used to extend the *servicemanager* daemon process to accept the permissions reported from application framework. They also adapt the *get_service*functionality to compute capabilities of the caller process and return it to the Binder driver.

Changes in the kernel are straightforward. They attach capabilities to the invocation requests. They also assign capabilities to new handles upon capability acquiring and delegation. A simple logic based on the access rights of the source and target takes place to decide if the delegation request is valid.

Part of the reasons behind the mass adoption of Android is its high usability from end user's perspective. Our design maintains this usability intact. In fact, end users cannot tell whether they are using the system in the capability or the normal mode.

TABLE 4.3: Newly Introduced Methods

Method	Return	Description
getSystemService(String name)	Binder handle	In the capability mode, this function causes the CM to compute access rights of the caller over the service, referenced by the name. The Binder driver then injects the access rights in the Binder reference and return the handle that references the Binder reference to the caller process. The application framework builds a proxy around that handle. The proxy then gets wrapped by a manger. In both operating modes, this API returns a manager.
getAccessRights()	Integer	Used on the service manager object to retrieve the access rights associated with the capability. This helps the developer to decide what access rights can be delegated.
writeDelegatedStrongBinder (IBinder binder, int delegated-Capability)	void	Used to delegate a capability to a process that listens on the other side. This API is called on the parcel sent to the target process. The same API is used to revoke access to a capability by setting the delegatedCapability to 0.

Chapter 5

Security Analysis

In this section, we analyze the new security features introduced by our design. We first start by laying out our assumptions concerning Android's permission system and Binder framework. Then we present the attacker model illustrating attackers capabilities and goals. Later, we discuss known attack scenarios against stock Android and shed light on how we can employ the new security features to prevent those attacks. We also present other attack scenarios introduced, or became more easier to establish, by our implementation.

5.1 Assumptions

To reasonably argue about the benefits and the shortcomings of our design from a security point of view, we need to establish a few assumptions that remain true throughout this discussion.

First, we assume that the Binder framework guarantees the unforgeability and uniqueness of Binder handles. Thereby, Binder capabilities are also guaranteed to be unforgeable and unique for each process. We further assume that data transferred over Binder IPC is confidential and tamper-proof. Thus, we roll out root-based attacks against the Binder framework [27] or even the kernel [52][55] from our consideration. Such attacks have devastating impacts on the whole system and would trivially break our assumptions. We also exclude attacks that aim to jeopardize system functionality by misusing the Binder framework, i.e., the DoS attack described by Huan Feng et al. [35]. Additionally, we assume system services and their host processes are immune to attacks and implement bug-free access control (this assumption rolls out the attack against *system_server* process [3]).

We further assume that Androids permission system cannot be bypassed. This means all installed apps, either through app managers or ADB, would have to declare required permissions in their manifest files. Permissions will be granted to the apps only upon the consent of the end user, either at runtime or install time. Therefore, Attacks that trick end user to grant permissions to malicious apps using overlay UIs [54] and phishing techniques are also excluded from our analysis.

In general, we focus on attacks that circumvent the permission system or take advantage of the Binder framework, i.e., for communication and remote code execution, to cause privilege escalation and violate the principle of least privilege.

5.2 Attacker Model

The attacker in our case is capable of writing malicious code and hiding it inside apps and 3^{rd}-party libraries. Such malicious code will eventually run on the mobile devices of the end users, i.e., by installing the malicious apps or including the malicious libraries in other installed benign apps. We assume the attacker has multiple malicious apps owned by her that could run on user's device. For simplicity, we disregard the case of running multiple benign apps that include different malicious libraries owned by attacker as it would have the same effect of running two malicious apps owned by the attacker.

The ultimate goal of the attacker is to leak protected sensitive information of end users without risking of being discovered and getting labeled as malicious. Otherwise, end users will not install her malicious apps, and other app developers will not include her malicious libraries in their benign apps.

We also assume the attacker to be smart and not to request permissions that deemed unreasonable to end users based on the announced functionalities of the apps or libraries.

We further assume the attacker is aware of the continuous endeavors of app developers and end users to limit her privileges on devices resources. Therefore, we assume the attacker would try to circumvent those measures to achieve her goals. This is essential when we discuss our approach for sandboxing malicious 3^{rd}-party library using capabilities.

Finally, we assume the attacker to be knowledgeable of other apps running on the system and expert enough to discover and exploit exposed services of those apps. This assumption is a necessary to launch confused deputy attacks.

5.3 Attack Scenarios

In the following, we present two categories of attacks. The first category consists of attacks that can be prevented or mitigated by our design. The second category includes attacks that have been introduced or became, even more, easier to establish in our design.

5.3.1 Mitigated Attacks

In this subsection, we present two attack scenarios which can be mitigated by our design.

5.3.1.1 Confused deputy

In this attack scenario, a benign app (called a deputy) is tricked into executing a sensitive operation on attackers behalf who does not have the required permissions to execute the operation herself. Androids security model cannot prevent this type of privilege escalation because it cannot reveal the identities of the apps involved in the transitive invocations.

Several solutions have been proposed to mitigate this attack [43][34]. Michael Dietz Wu et al. [31] proposed to keep track of the call chain enabling apps to authenticate the chain at runtime and dropping the call in case one app does not have the required permission. Another work by Bugiel et al. [30] which also relies on the call chain, provides a reference monitor that enforces transitive policies on IPC between apps and system services.

We mitigate this attack using capabilities. However, our solution assumes that the app developer has some degree of security awareness and is willing to invest more efforts in hardening her app. We further assume that the deputy is deliberately exposing an interface to other processes. This interface makes a call to a protected operation from a system service. Only processes possessing a specific access right/permission can execute this operation through the interface.

To illustrate our approach for mitigating the confused deputy problem, we present Figure 5.1 which depicts three processes, i.e., (A), (B), and (C), and a service (S). Process (A) is under the control of the app developer. It exposes an interface that implements a complex functionality which invokes the operation (O) from the service (S). The service (S) requires processes willing to call operation (O) to possess a capability to it with access rights of (0x1). Processes (B) and (C) hold capabilities to (S) with access rights of 0x0 and 0x3 accordingly. The goal is to permit (C) to access operation (O) through (A)'s interface while preventing the same operation for (B).

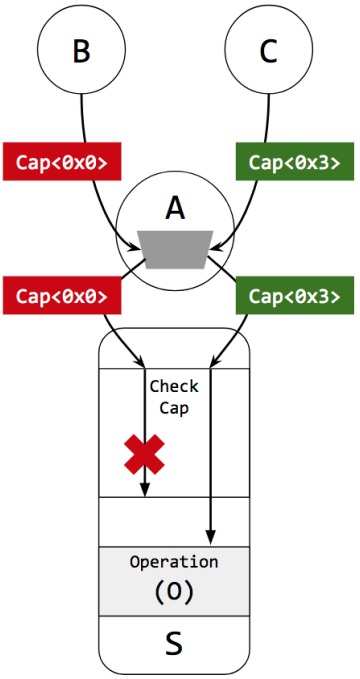


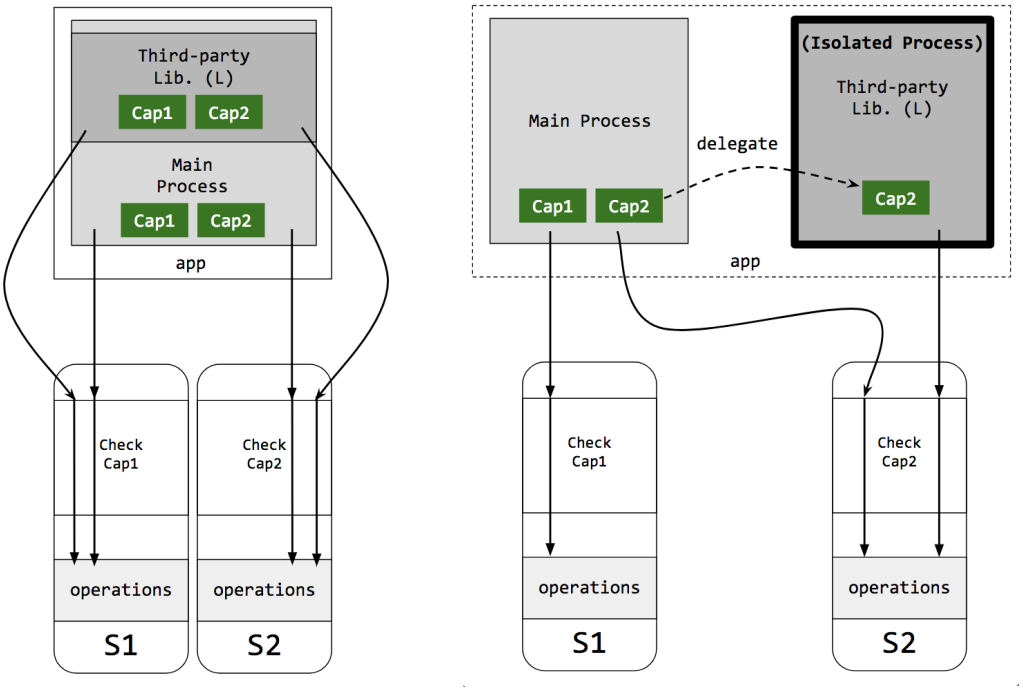
FIGURE 5.1: Mitigating The Confused Deputy Attack

We can achieve the aforementioned requirement through capability delegation. Both processes (B) and (C) have to delegate their capabilities, for service (S), to process (A). In turn, process (A) will use the delegated capabilities to invoke the operation (O). Notice that (A) does not have to enforce any type of access control as it moves this task to the service. Consequently, the service will allow process (C) to access the service, as it has the required access rights, and reject deny the request from (B).

5.3.1.2 Inclusion of Malicious Library

Developers rely on 3^{rd}-party libraries for reducing the programming effort, providing analytics, and monetizing apps. All included 3^{rd}-party libraries run inside apps sandbox and share the same privileges granted to the app. Additionally, such libraries tend to request even more permissions putting users privacy and security at risk of being compromised [28][42][36]. Based on our attack model, we assume that 3^{rd}-party libraries can be malicious by themselves and must not run with the full authority of the parent sandbox. Instead, 3^{rd}-party libraries must be granted the least privileges they require and deemed to be reasonable to app developers.

Several approaches have been proposed to address this issue and they vary from completely blocking 3^{rd}-party libraries [53], especially advertisements libraries, to compartmentalizing them [44][56]. One novel approach that does not break the same origin policy of apps nor requires firmware changes is what Jie Huang et al. [38] have proposed



(A) Library Runs In The Main Process (B) Library Runs In An Isolated Process

FIGURE 5.2: Mitigating The Problem of 3^{rd}-party Library

of isolating the advertisements library into a separate app at compile time and assign it the minimum permissions required. As a result, new Binder IPC protocol needs to be established between the original app and the app which runs the library.

Our design addresses this problem through delegation of capabilities. Before presenting the approach, we setup a scenario of the problem and define our goals. As depicted in Figure 5.2a, we consider a case of an app (A) that runs a 3^{rd}-party library (L). There are two system services (S1) and (S2) which are protected by (Cap1) and (Cap2), accordingly. These capabilities are granted to the app (A). We assume that library (L) requires the (Cap1) to access the service (S1) to perform some functionality. Since the library (L) runs inside the sandbox of the app (A), it can access both services (S1) and (S2). The goal is to prevent that from happening and only allowing the library (L) to access the service (S1).

Our proposed solution requires the app developer to be effectively involved in the process of hardening the app. According to the attacker model, we assume that the developer of the library is willing to cooperate with app developers so that she would look honest while hiding her malicious intentions. As depicted in Figure 5.2b, our solution constitutes that library (L) must run in an isolated process, while the rest of the app runs in the main process. Isolated processes are prevented from accessing any resource because they can not acquire permissions nor possess capabilities directly from the CM. The main process would acquire two capabilities to (Cap1) and (Cap2) and only delegate

the capability for (S1) to the isolated process. This is technically possible as no SELinux policies are enforced to prevent the Binder driver from transmitting Binder handles (and capabilities) to isolated processes. The access rights of the delegated capability must be instrumented to only grant the least privilege required by the library to function properly.

As mentioned earlier, this approach requires cooperation from the developers of 3^{rd}-party libraries as they need to implement special interfaces to receive the delegated capabilities. They additionally need to adapt their code to use the delegated capabilities instead of requesting their own (which will not be possible).

5.3.2 Attacks Against Our Design

In this subsection, we present two types of attacks that have been introduced or became more easier to establish due to our implementation.

5.3.2.1 Collude attacks

Similar to the confused deputy attack, collude attacks benefit from the shortcomings of Androids permission system and its inability to detect and enforce policies on transitive invocations. Colluding apps normally belong to the same attacker and use overt and covert channels for communications between them. The challenge, from attackers perspective, is how to employ two or more apps that acquire different permissions to collaboratively leak sensitive information about the end users.

For example, one app, called (A), has permission to users phone book but does not have permission to use the internet. Another app, called (B), can access the internet. Both apps are under attackers control. The collude attack can be easily established by retrieving users phone book in the app (A), sending it to the app (B) which, in turn, sends it to a remote server breaking end users privacy.

This attack becomes more severe in our design because colluding apps can communicate capabilities between each other causing privilege escalation for all of them. The countermeasure is to prevent delegation among apps and limit it to processes of the same app. Our design does not yet implement this mitigation technique. In *Chapter 7: Future Work*, we present an idea on how we can implement this extension to prevent this type of attacks.

5.3.2.2 Overwriting Access Rights

This attack is directly introduced by our design due to the simplified implementation of the delegation process we have decided to adopt. To understand the attack, we consider a hypothetical scenario of a malicious process (A) that wants to delegate a capability for a system service (S) to a benign process (B). Assuming both processes reside on different app sandboxes. The current implementation enables process (A) to downgrade the access rights of all the capabilities possessed by the process (B).

Process (A) establishes the attack by creating capabilities to all services in the system, including service (S). Then, delegate those capabilities to process (B) with zero access rights. Since the Binder driver keeps only one capability for each system service, the easiest way to realize the delegation of capabilities is through overwriting the access rights while keeping the references of capabilities intact. As a result, process (B) would have capabilities to all system services with zero access rights on them.

This attack becomes meaningless if delegation among apps is prohibited, or if process (B) does not accept delegation (e.g., it does not expose interfaces for delegation). Moreover, process (B) can overcome this attack by invalidating old capabilities and requesting new ones by calling *getSystemService()*. This guarantees that the CM will compute the most up-to-date access rights and overwrite the value delegated from process (A).

Chapter 6

Discussion and Evaluation

In this chapter, we evaluate our design in three aspects: Performance (to see how much overhead/gain our design introduces), coverage and effectiveness (by evaluating how close our implementation is to enable capabilities for all system services), and usability (to check whether our changes would affect developer's and user's experience while using the system or developing apps). We end the chapter by holding a discussion on whether it is possible to utilize SELinux to achieve the same outcomes of our design.

6.1 Performance Analysis

In this section, we conduct three time-measurement experiments that show the performance gain and overhead caused by our design. We start by presenting the setup for those experiments, then we explain, in details, how they are conducted before presenting the results.

6.1.1 Configurations and Setup

Table 6.1 shows versions and configurations used to build the AOSP and the Goldfish kernel. For some experiments, we built each project twice, one build includes our changes that implement our approach, and the other build is kept without any modifications (except for the time measurement code and testing apps). We used the default ARM Android emulator shipped with AOSP for all experiments.

Table 6.1: AOSP and Kernel Build Information

AOSP	version	7.1.2
	branch	android-7.1.2_r33
	target build	full-eng
Goldfish Kernel	version	3.4
	config	goldfish_armv7_defconfig

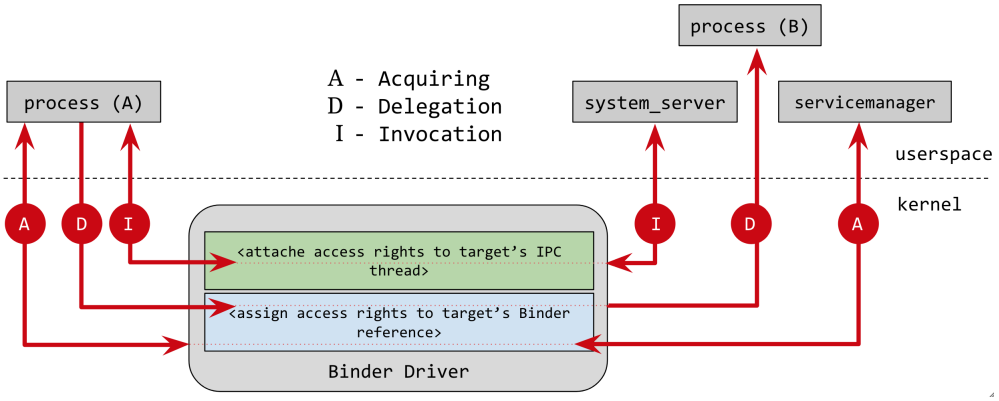


Figure 6.1: Time Measurement Components

6.1.2 Experiments and Results

Our goal in these experiments is to measure the execution time for three operations: capability acquiring, delegation, and invocation. Given that delegation and revocation of capabilities use the same technique, both operations have exactly the same execution time, and it is adequate to only measure one of them.

To accurately measure the aforementioned operations, we need to isolate them from other operations that might affect the results. For example, we cannot measure the invocation time from the client side by simply capturing the time before the invocation is made and after the result is back, and then subtract both measurements from each other. This is because the results would highly depend on the workload on the system. For example, it is possible that the Binder driver delays the invocation request because it handles other IPC requests that have higher priorities or simply arrived earlier. Since we cannot guarantee that all requests will be handled on-time, without delays, we have to find another solution.

We can assume that the cost paid while the request is traveling from one endpoint to the Binder driver, and from the Binder driver to the target endpoint remains constant for all IPC requests. This travel time (TT) appears as single-pointed red arrow in Figure 6.1. The double-pointed arrows denote double of the TT, e.g., acquiring a capability costs $4{\times}TT$. This assumption simplifies the measurements as we only have to measure the

operations that happen at the endpoints and in the Binder driver. Our design introduces two operations that take place at the endpoints. Those operations are: Encoding the access rights from the local permissions stored in the using *compute_capability()* in the CM [1] (for the operation of acquiring capabilities) and performing access control on system services using capabilities (for capability invocation). The operations that happen at the Binder driver are: attaching the capability to the request (for invocation and delegation) and saving the capability in a Binder reference (for capability acquiring and delegation).

As a result of this simplification, we can measure the time needed for acquiring a capability as following: $4\times TT$ + time(compute_capability) + time(assigning access rights to a Binder reference). Notice that Figure 6.1 shows the point of time when the Binder gets involved in the operation (e.g., when the red arrow touches one of the two boxes in the Binder driver)

Measurements At The Context Manager

In stock Android, the *getSystemService()* API caches all live Binder handles retrieved through it. This is meant to reduce the number of IPC requests made to the CM and, consequently, improve performance. For the sake of this experiment, we disabled the caching to make sure that all requests reach the CM.

Our goal in this experiment is to measure the execution time of the function *compute_capability()* from *service_manager.c*. We conducted the experiment by running three apps that request all ten normal and dangerous permissions to the four different system services (location, wifi, connectivity, and wallpaper). When permissions are granted, each application requests a capability for each system service. In total, the *compute_capability()* function was executed about 1000 times. Since permissions are stored in a linked list, we chose the ten requested permissions carefully to trigger traversal of almost all nodes in the permissions list. In total, the list contained about 180 permissions for 41 apps.

This experiment shows that the average time of computing access rights is 1.7ms with a standard deviation of 2.1ms. The time measurements were done using *gettimeofday()* function from <*sys/time.h*>. The experiment shows a pressing need for re-implementing the *compute_capability()* function and associated data structures to reduce the time and storage overhead.

[1] Since permissions change infrequently, we neglect the overhead caused by reporting permissions from the PMS to the CM because the reporting is done over Binder IPC and carry parameters of primitive types. Therefore, reporting a permission costs only $2\times TT$ since it is one-way Binder IPC

Measurements At System Services

In our design, system services perform in-place access control using cheap bitwise arithmetic operations on the access rights associated with the capabilities of calling processes. This comes with a performance gain as services of stock Android rely on the PMS to check for caller's authorization for each invocation to the protected methods.

We conducted the experiment by build two images of AOSP. One image uses the conventional reference monitoring using the PMS, whereas the other includes our changes and implements an access control based on capabilities instead of using the PMS. Each image includes an app that requests the ACCESS_FINE_LOCATION permission. When the permission is granted by the user, both apps acquire a capability to the LMS and call the *getLastLocation()*. We use *System.nanoTime()* to measure the execution time of the permission/access rights checking in both images.

After invoking the reference monitor of both implementation a 1000 time, the results show that our approach outperforms the conventional permission checking by a factor of approximately 4.5. The average time overhead caused by our approach is about 20µs and with a standard deviation of approximately 15.5µs. On the other side, the average execution time of the permission check logic is about 87µs with a standard deviation of about 153µs.

It worth noting the LMS that hosts the *getLastLocation()* resides in the same process of the PMS (which is the *system_server* as discussed earlier). This causes a local call to the Binder object of the PMS for permission check. We believe that the amount of time required for checking camera permissions (from *mediaserver* process) would show a significant increase in the execution time as the check would be performed over an IPC. However, we did not make this experiment for time limitations.

Measurements At The Binder Driver

As shown in the Figure 6.1 and following our discussion on *Section 4.3 Capability Management*, the Binder is extended to support two tasks: First, it attaches the access rights of the calling process to the IPC thread that carries the transaction data to the system service (see the light green rectangle in Figure 6.1). Second, when a process delegates a capability to another process or when the CM returns a capability to a process that requested it, the Binder driver assigns the access rights of the capability to the Binder reference that is newly created for the receiver process (see the light blue rectangle in Figure 6.1). Notice that the Binder driver handles the capability returned from the CM to the requester process as a special form of delegation. Therefore, it is handled with the same function used to validate delegation requests.

The first task does not produce any performance overhead, in comparison to stock Android, as it is exactly similar to how the Binder attaches caller's UID and PID to the IPC thread of the system server. However, the second task yields on average an overhead of about 2.69µs with a standard deviation of about 1.47µs. This overhead is caused by the logic used for deciding if the delegation is valid. We have derived those results by conducting the following experiment: Process (A) delegates the capability (for LMS) to the process (B). This operation is repeated a 1000 time with different access rights to make sure that only 50% of the delegation requests comply with the delegation rules and, therefore, are successful.

6.2 Coverage and Effectiveness

During the development phase and while we were examining the possibility for applying capabilities on all system services (so access to system services can be delegated, revoked, and enforced by the access rights of capabilities) we came across five categories of permissions that differ based on the time and place of enforcement.

1. Permissions enforced by system services by calling protected methods using service managers: This type of permissions is the focus of this work. Although the permission check could happen in the native code (e.g., CameraService.cpp [16] enforces the permission CAMERA), enforcement of this type of permissions can easily be replaced by capability checks and access to system services (for almost all permissions) can be delegated and revoked. See Table 6.2 for a complete list of permissions for this category.

2. Permissions enforced by content providers[2]: Content providers manage access to a repository of data. Developers can access this data through a ContentResolver object that is retrieved from app's context. The ContentResolver composes a handle for the ContentProvider, which is retrieved through the CM. The ContentResolver can be used to perform CRUD operations on contacts, calendar, and SMS data. Developers specify a URI to the repository they wish to access and the ContentProvider makes two authorization checks. The first is made to the PMS to decide if the caller has read or write permissions (e.g., READ_CONTACTS and WRITE_CONTACTS). The second is made to the AMS to decide if the app has access to the data source (referenced by its URI). Although we can use capabilities to replace the first check, we cannot get rid of the second check. The AMS keeps a database of all packages

[2]We only consider content providers offered by the system, e.g., to access the contacts, calendar, and SMS messages.

and URIs assigned to them[3], this database is modified only when an app is granted a permission by the user to a specific resource. Consequently, if we delegated access to content providers to other processes, the first check could pass (as it would rely on the access rights of the delegated capability). However, the second check will always fail because the receiver did not explicitly acquire the permission.

3. Permissions enforced by the AMS upon delivering broadcast messages: Most permissions are enforced when apps make interactions with the system, e.g., a background service invokes a protected method of a system service, a user clicks a button that retrieves data from system services, etc.). However, some permissions are enforced when the system issues broadcast messages. For example, when an SMS message is received or when the system finishes booting up. Those broadcasts are issued by the AMS which checks the permissions of each app to decide whether to deliver the broadcast message to it or not. The current implementation does not provide a solution on how to make the authorization check based on capabilities.

4. Permissions enforced by system services when apps send broadcasts or start activities using Intents: In the current implementation, enforcement of this type of permissions cannot be replaced by capabilities.

5. Permissions mapped to GIDs and enforced by the kernel.

For the complete list of permissions that are not supported by our design, see Table 6.3. Notice that permissions are associated with a category that corresponds to one of the five categories above.

The percentage of permissions that can be enforced using capabilities is about 60%. We support eight dangerous permissions out of 20 defined in the system. As a proof of concept, we believe we have achieved good coverage. However, the current approach is far away from being complete.

[3]This facilitates sharing data between apps. For example, an app can allow another app to access a specific record but not the whole database by exposing a URI to that record.

Table 6.2: Permissions Can Be Enforced By Capabilities

Name	**Level**	**Manager**
ACCESS_FINE_LOCATION	dangerous	LocationManager
ACCESS_COARSE_LOCATION	dangerous	LocationManager
SEND_SMS	dangerous	SmsManager
USE_SIP	dangerous	SipManager
READ_PHONE_STATE	dangerous	TelecomManager
CALL_PHONE	dangerous	TelecomManager
(*) BODY_SENSORS	dangerous	SensorManager
(*) CAMERA	dangerous	CameraManager
READ_SYNC_SETTINGS	normal	ContentResolver
READ_SYNC_STATS	normal	ContentResolver
WRITE_SYNC_SETTINGS	normal	ContentResolver
BROADCAST_STICKY	normal	ActivityManager
KILL_BACKGROUND_PROCESSES	normal	ActivityManager
REORDER_TASKS	normal	ActivityManager
SET_TIME_ZONE	normal	AlarmManager
MODIFY_AUDIO_SETTINGS	normal	AudioManager
ACCESS_NETWORK_STATE	normal	ConnectivityManager
CHANGE_NETWORK_STATE	normal	ConnectivityManager
TRANSMIT_IR	normal	ConsumerIrManager
USE_FINGERPRINT	normal	FingerprintManager
DISABLE_KEYGUARD	normal	KeyguardManager
ACCESS_LOCATION_EXTRA_COMMANDS	normal	LocationManager
ACCESS_NOTIFICATION_POLICY	normal	NotificationManager
WAKE_LOCK	normal	PowerManager
EXPAND_STATUS_BAR	normal	StatusBarManager
VIBRATE	normal	Vibrator
SET_WALLPAPER	normal	WallpaperManager
SET_WALLPAPER_HINTS	normal	WallpaperManager
ACCESS_WIFI_STATE	normal	WifiManager
CHG_WIFI_MULTICAST_STATE	normal	WifiManager
CHANGE_WIFI_STATE	normal	WifiManager
(**) GET_PACKAGE_SIZE	normal	PackageManager
(***) BLUETOOTH	normal	BluetoothManager
(***) BLUETOOTH_ADMIN	normal	BluetoothManager

* Permission check happens in native code

** We could not successfully extract the handle out of the PackageManager to delegate it. Therefore, we assume that access to the PMS cannot be delegated.

*** Permission is also mapped to GID and enforced by DAC and MAC.

Table 6.3: Permissions That Is Not Supported By Our Design

Category	Name	Level
2	ADD_VOICEMAIL	dangerous
2	READ_CALL_LOG	dangerous
2	WRITE_CALL_LOG	dangerous
2	READ_CONTACTS	dangerous
2	WRITE_CALENDAR	dangerous
2	WRITE_CONTACTS	dangerous
2	READ_CALENDAR	dangerous
2	READ_SMS	dangerous
3	PROCESS_OUTGOING_CALLS	dangerous
3	RECEIVE_SMS	dangerous
3	RECEIVE_WAP_PUSH	dangerous
3	RECEIVE_MMS	dangerous
3	RECEIVE_BOOT_COMPLETED	normal
4	INSTALL_SHORTCUT	normal
4	UNINSTALL_SHORTCUT	normal
4	REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	normal
4	SET_ALARM	normal
4	REQUEST_INSTALL_PACKAGES	normal
5	READ_EXTERNAL_STORAGE	dangerous
5	WRITE_EXTERNAL_STORAGE	dangerous
5	INTERNET	normal
unknown	RECORD_AUDIO	dangerous
unknown	NFC	normal

6.3 SELinux vs. Capabilities

Capabilities introduce new security concepts that exist in neither DAC nor MAC. Specifically, capabilities define fine-grained access rights on resources. A process that holds a capability can delegate access to other processes and then revoke it. Ideally, the system can also upgrade and downgrade access rights of issued capabilities without interrupting processes or causing them to restart.

Given that SELinux, as a MAC, was introduced to cover the shortcomings of DAC by defining fine-grained access rights through policies, one interesting question is raised: Is it possible for SELinux to provide the same security features which are enabled by capabilities? We answer this question using the existing functionalities offered by SELinux framework as implemented in Android.

First, we start with a simple case where Android, hypothetically, has only one object which provides N functionalities that need to be protected. Using capabilities, each process acquires a unique capability token that has N bits which encode access rights on

the N functionalities of the object. The object, in turn, implements access control by performing bit-masking on the access rights of the capabilities associated with callers.

One way to establish the same security enforcement using SELinux is to create 2^N types[4] which encode every variation of the N permissions. Each process will be assigned a security context with the type that corresponds to the permissions it acquires. Creating this amount of types is inevitable because we do not know what permissions each process will exactly acquire during its lifespan. For example, a process might start with zero permissions, then acquire more permissions at runtime (as app processes in Android which are granted permissions by the end user at runtime). Therefore, all possible variations of the N permissions must be created beforehand. The object, on the other hand, must be assigned a type that never changes[5].

For each of the 2^N types, a rule must be defined to govern the functions which can be accessed for each type. The Listing 6.1 shows an example of the rules that need to be created for an object that offers only two functionalities (func1, and func2). One important observation from that listing is that no rule has to be defined to prevent an unprivileged process from accessing the object because the absence of such rule is interpreted as a prevention by default. The first rule means that processes of the type "perm1_1" can execute both functions of the object that has the type "object_type" and of class "object_class"

```
1 allow      perm1_1      object_type:object_class      *;
2 allow      perm0_1      object_type:object_class      func2;
3 allow      perm1_0      object_type:object_class      func1;
```

Listing 6.1: SELinux Allow Rules

So far, one important question is unanswered; Who assigns types to processes in Android, and when? By default, forked process inherits the security context of the parent process. To enable switching between security contexts, SELinux enables privileged processes to change types of self and others, using API exposed from SELinux framework. For example, *Zygote* executes *selinux_android_setcontext()* on newly forked processes to change their security contexts, and init.c calls *setcon()* to change its type.[33]. Another way for changing security contexts is through transition rules (type_transition).

[4]SELinux is for whitelisting accesses. Therefore, the absence of a rule implies prevention. Consequently, the number of types and rules is, in fact, 2^N - 1 because processes that have no access rights can be assigned a type that is not associated with any rule.

[5]As discussed in *Subsection 2.2.1 Application Sandboxing*, rules rely on the types of subjects and objects. Changing object's type would prevent access to it unless other rules with the new type are defined

Transition rules are checked when a process executes a special file. The execution, if allowed, causes the security context of the caller process to change. The new security context is defined based on the original type and the type of the file executed by the process. The file represents an entry point to a type transition (this is defined in a rule), and it is protected by a permission. The process can only cause a change of security context if it is allowed by another rule to execute the file (granted the permission). Therefore, for each type transition policy, three rules need to be defined, and one executable file should be created and assigned a type [26]. Next, we need to employ one of these techniques to grant permissions, at runtime, to processes.

Since policies should be created, compiled, and attached to Android images before distribution, the system administrator must create $2^N \times (2^N\text{-}1)$ executable files (and the same number of types assigned to those file) and triple that number of rules to define transition policies that enable a process to move from any of the 2^N security contexts to any other context realizing privilege escalation or degrading[6]. To prevent processes from illegitimately escalate their privileges, the system[7] exposes the executable file to be executed by the process only when a change of permission associated with the process occurs. The problem resides on how to force a process to execute a file that degrades its privileges assuming the process is curious or malicious and would not do that by itself. Apparently, this approach can easily scale up to become impractical to implement. Therefore, it is not considered as a solution.

This leaves us with the APIs exposed by SELinux framework. In fact, this approach deems to be more reasonable. However, the process that causes a change of security context of another process must be 1) trusted to prevent unintended privilege escalation, and 2) able to accurately compute the right security contexts for the processes that reflect permissions acquired by them at runtime. In Android middleware, *Zygote* can be modified to fulfill the second requirement and, therefore, protect system services similar to what we have done in this work. For example *Zygote* gets notified whenever a permission is granted/revoked to/from an app and then, it changes the type of running processes of that app; or when a new process is started, it queries the permissions associated with its app and assigns the new process to the type accordingly. Since permission change at runtime is specific to app processes, this solution could work out.

[6] Another solution could be to create an intermediary type that all processes willing to change types have to go through it. This reduces the number of executable file to 2×2^N. For example, to change type X to Z, the process has to change its type to Y first. Thus, transition rules allowing moving from X to Y and, then from Y to Z are required. However, this assumes the system can hide the files that allow a process of type Y to move into Z" context, where Z" has more privileges than Z.

[7] The process that decides what to expose/hide should have a notion of the permissions assigned each process has and should next have.

The discussion so far considers Android with only one service to protect. However, in practice, Android offers a huge number of objects that need to be protected. We have introduced 2^N types for a single object that offer N functionalities. When a process gets assigned a type from those types, it can access only that object with respect to the encoded permission in the type. However, processes should be able to have access to multiple types simultaneously. Given that a process can only have one type and rules define a relation between types of subjects and objects. All objects must have the same type. Therefore, the sum of all permissions offered by all objects in the system is N'. Considering that the number of types and rules is $2^{N'}$, the system could easily become impractical to implement, if not even impossible.

Chapter 7

Future Work

In this section, we present a few ideas on how to push current design towards the adoption of capabilities globally. We also discuss a few modifications that are necessary to harden the security of our design.

Global Capabilities

To support global capabilities, we have to address three challenges:

First Challenge: the first challenge would be to sandbox all processes that issue direct system calls to the kernel (this includes system services and daemons) and only grant them the least access rights needed. One conceivable solution would be to borrow the concept of UNIX capabilities from Capsicum project and apply it at this low-level. However, given the huge number of daemons and server processes, the real challenge would be to 1) decide exactly what access rights each process needs, and 2) get to know the system calls that have to be capability-aware.

Second Challenge: When app developers create bounded services, they normally define permissions to protected their functionalities. Other apps willing to use this service, have to acquire the necessary permissions. The service, in turn, would act as a reference monitor that uses the PMS to query whether the calling process has the required permissions or not. Although user-defined services are accessible over Binder IPC, they are not registered in the CM. As a result, they are not supported by our design. The flow on how processes acquire a Binder handle to a user-defined service is yet to be discovered. If we managed to encode user-defined permissions, defined for the service, into a mask of access rights and attach it to the Binder handle of the bounded service, we would solve the second challenge towards applying global capabilities.

Third Challenge: Some highly privileged processes, such as SurfaceFlinger, run in the "system" sandbox with UID of 1000. Such services require some functionalities from other system services registered with the CM. System services are designed such that they allow calls that originate from processes with specific UIDs (e.g., 1000, indicating that it is a system process) to pass without permission check. This, in fact, is the third challenge. One conceivable solution to this issue would be that system processes have to identify themselves (using an identifier other than UID) when calling the CM asking for a capability to a system service. The CM, in turn, have to maintain a static list of access rights for each system process to all required system services. In turn, system services will enforce access control using capabilities for all requests, including requests from system processes.

Finally, it worth noting that, addressing the aforementioned three challenges on the current design leads to complete abandon of UIDs. This means the ambient authority would no longer exist.

Security Enhancements

The current design allows delegation among apps. This makes it easy for malicious app developers to collude with other apps of her own to sabotage users privacy and security. The solution to this problem would be to only allow delegation among processes of the same app. We have two conceivable paths that can be implemented in the Binder driver; First, upon capability delegation, abort the operation if the sender and receiver have two different UIDs. Second, introduce SELinux policies and hooks in the Binder to prevent delegation between processes of different types. The first solution is undesirable given that we are trying to eliminate the reliance on UIDs in the system even though it might be the easiest to implement.

Finally, although defining fine-grained access rights on system services may lead to confusion for end users and reduce usability, it would undoubtedly improve security. However, we can leave the Android's permissions intact while empowering app developers by defining more fine-grained access rights on system services which are enforced only when the calling capability is delegated. For example, a service provides two sensitive methods protected by a permission. In the capability mode, have performs two types of access control. First, when a request with a capability issued by the system arrives, it would allow execution of both methods if the first bit of access rights is set. The second case is when a request with a delegated capability[1] arrives, then it would allow execution of both methods only if the first two bits are set. This way, the app developer

[1]The kernel can be extended to provide this information

can delegate access to individual methods to other processes, which will not be able to access other methods not included in the delegated capability.

Chapter 8

Conclusion

In this thesis, we presented our approach for supporting capabilities in Androids middleware. The approach extends Androids security model with new security features and gets the overall system a little bit closer towards applying the principle of least privilege.

The extension we have presented allows the system to work in two modes: normal mode, and capability mode. In the capability mode, conventional high-level permission enforcement on system services is disabled and access control based on capabilities becomes effective. Thus, the capability mode eliminates the effect of ambient authority and services no longer care on who is calling. Instead, they check whether the request has the required access rights or not.

We used the Binder framework as the core building block for capabilities. Based on that, we showed how to extend Binder handles with access rights to form fully-fledged capabilities. Then, we illustrated how Binder capabilities are used in service invocation, delegation, and revocation. The rationale behind using Binder handles to build capabilities resides in the fact that Binder handles are unforgeable and communicable tokens of authority. Those characteristics are the minimum requirements that must hold true for capability tokens.

We further discussed how using the Binder framework limits the scope of capabilities to only resources that are accessible through the CM. However, even with the presence of this limitation, we managed to address two issues: The confused deputy attack, and inclusion of malicious 3^{rd}-party libraries. We showed how app developers need to be more security-aware as they are given more power to limit the privileges of untrusted components of their apps. However, we also discussed how our design aggravates the effect of collude attack and open the door for new attack caused by implementational limitation. Nevertheless, we presented our conception on how to prevent those attacks.

Finally, we performed a performance analysis and showed that the overhead introduced by creating capabilities is hardly observed. We further showed how the in-place access rights check outperforms the conventional permission checking in PackageManagerService. We argued whether SELinux can achieve the same objectives accomplished by our approach and showed it might be impractical, or even infeasible, for SELinux to do so.

In conclusion, hybrid systems (which rely on DAC, MAC, and capabilities, such as Capsicum and our proposed design) reduce the effect of ambient authority, and show high potential in confining sandboxes and effectively applying the principle of least privilege.

Bibliography

[1] Android: Celebrating a big milestone together with you. https://blog.google/products/android/2bn-milestone/.

[2] Android documentation - app manifest. https://developer.android.com/guide/topics/manifest/manifest-element.html#uid.

[3] Bitunmap: Attacking android ashmem. https://googleprojectzero.blogspot.de/2016/12/bitunmap-attacking-android-ashmem.html. Author: Ben, Last check 27.12.2017.

[4] Github - system services retrievable by third-party apps. https://gist.github.com/abdawoud/630c4ecaca35cdb419b41249fb453def.

[5] Google. android developers - overview on android memory management. https://developer.android.com/topic/performance/memory-overview.html.

[6] Google. android developers - platform architecture. https://developer.android.com/guide/platform/index.html.

[7] Google. android source - android open source project. https://source.android.com/setup.

[8] Google. android source - application context. https://developer.android.com/reference/android/content/Context.html.

[9] Google. android source - hal architecture. https://source.android.com/devices/architecture.

[10] Google. android source - jack build toolchain. https://source.android.com/setup/jack.

[11] Google. android source - media server. https://source.android.com/devices/media/framework-hardening#mediaserver-changes.

[12] Google. android source - multiple users. https://source.android.com/devices/tech/admin/multi-user.

[13] Google. android source - storage permissions. https://source.android.com/devices/storage/.

[14] Google. android source code. https://android.googlesource.com/platform/frameworks/base/+/android-7.1.2_r33/data/etc/platform.xml.

[15] Google. android source code - bluetooth system service. https://android.googlesource.com/platform/frameworks/base/+/android-7.1.2_r33/services/core/java/com/android/server/BluetoothManagerService.java.

[16] Google. android source code - cameraservice. https://android.googlesource.com/platform/frameworks/av/+/android-7.1.2_r33/services/camera/libcameraservice/CameraService.cpp.

[17] Google. android source code - goldfish kernel. https://android.googlesource.com/kernel/goldfish.

[18] Google. android source code - install daemon. https://android.googlesource.com/platform/frameworks/native/+/android-7.1.2_r33/cmds/installd/installd.cpp.

[19] Google. android source code - libcore-io-posix library. https://android.googlesource.com/platform/libcore/+/android-7.1.2_r33/luni/src/main/native/libcore_io_Posix.cpp.

[20] Google. android source code - system server. https://android.googlesource.com/platform/frameworks/base/+/android-7.1.2_r33/services/java/com/android/server/SystemServer.java.

[21] Google. android source code - uids and gids of system daemon and services. https://android.googlesource.com/platform/system/core/+/android-7.1.2_r33/include/private/android_filesystem_config.h.

[22] Google. google source - selinux. https://source.android.com/security/selinux/.

[23] Google. permission levels. https://developer.android.com/guide/topics/manifest/permission-element.html#plevel.

[24] Google. security enhancements in android (runtime permissions). https://source.android.com/security/enhancements/enhancements60.

[25] Hal types, android source. https://source.android.com/devices/architecture/hal-types.

[26] Selinux - centos wiki. https://wiki.centos.org/HowTos/SELinux.

[27] Nitay Artenstein and Idan Revivo. Man in the binder: He who controls ipc, controls the droid. 2014.

[28] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. *CoRR*, abs/1303.0857, 2013.

[29] Sven Bugiel. *Establishing Mandatory Access Control on Android OS*. PhD thesis, Saarland University, 2015.

[30] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)*, Feb 2012.

[31] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[32] Xuhua Ding Zhoujun Li Dong Shen, Zhangkai Zhang and Robert H. Deng. H-binder: A hardened binder framework on android systems. 2016.

[33] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014.

[34] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[35] Huan Feng and Kang G. Shin. Understanding and defending the binder attack surface in android. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 398–409, New York, NY, USA, 2016. ACM.

[36] Xing Gao, Dachuan Liu, Haining Wang, and Kun Sun. Pmdroid: Permission supervision for android advertising. *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 120–129, 2015.

[37] Dianne Hackborn. Discussion on binder. `https://lkml.org/lkml/2009/6/25/3`.

[38] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *CCS*, 2017.

[39] Iliyan Malchev Keun Soo Yim and Dave Burke. A taste of android oreo (v8.0) device manufacturer. Technical report, Google, 2017.

[40] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[41] Sebastian Gerling Michael Backes, Sven Bugiel. Scippa: System-centric ipc provenance on android. 2014.

[42] Gabriel Núñez and Anthony D. Joseph. Party pooper: Third-party libraries in android. 2011.

[43] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[44] Paul Pearce, Adrienne Porter Felt, Gabriel Núñez, and David A. Wagner. Addroid: privilege separation for applications and advertisers in android. In *ASIACCS*, 2012.

[45] S. A. Rajunas, Norman Hardy, Allen C. Bomberger, William S. Frantz, and Charles R. Landau. Security in keykos. In *IEEE Symposium on Security and Privacy*, 1986.

[46] Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. A systematic security evaluation of android's multi-user framework. *CoRR*, abs/1410.7752, 2014.

[47] Thorsten Schreiber. Android binder - android interprocess communication, 2011.

[48] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *SOSP*, 1999.

[49] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. 2012.

[50] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, 2010.

[51] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[52] Wen Xu and Yubin Fu. Own your android! yet another universal root. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. USENIX Association.

[53] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps, 01 2017.

[54] Simon P. Chung Yanick Fratantonio, Chenxiong Qian and Wenke Lee. Cloak and dagger: From two permissions to complete control of the ui feedback loop. IEEE '17. ACM, 2017.

[55] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1093–1104. ACM, 2015.

[56] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: isolating advertisements from mobile applications in android. In *ACSAC*, 2013.

[57] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.