AmusePermit: Android 민감 API 의 권한 우회 및 부여 일원화를 위한 IPC 프레임워크

최유준 (Yu Jun Choi)
Hankuk University of Foreign Studies
cuj1559@gmail.com

June 14, 2024

Abstract

본 연구에서는, Android 시스템에서 민감한 API 의 앱 별 권한 부여를 우회하고 SDK 버전 별 구현 파편화를 일원화하기 위해, Binder IPC 의 Wrapper 구현 중 하나인 BroadcastReceiver 와 ContentResolver API를 혼합, 일종의 앱 간 핸드셰이크 및 데이터 교환 프로토콜을 구현하였으며, 이에 대해 IO Stream 레이턴시를 비교하는 등의 방법으로 Native 구현과 대비하여 본 프레임워크의 성능을 비교분석하였다.

Keywords: *Android, IPC, Binder, Security, BroadcastReceiver, ContentProvider, Privilege Escalation, Sandbox, Process isolation*

1 Introduction

구글은 최근 민감 API의 사용에 대한 규제 및 제한 사항을 점차 확대시킴에 따라 파일, SMS, 및 위치와 같은 민감한 API 정보 접근 앱 개발에 필요할 수 있는 기능들에 대한 권한 부여에 제한이 이루어지는 한편, Android의 SDK 버전이 점진적으로 업데이트 됨에 따라, 이들 민감 API 의 접근 권한 부여 구현 사양이 점진적으로 변화되면서 SDK버전 별 파편화가 확대되며 이들 API의 사용이 점차 어려워 지고 있으며, 이는 앱 개발 및 하위 호환성을 비롯한 유지보수에 큰 애로사항을 남기고 있다. 또한, 과거에는 IPC를 활용한 권한 상승의 방법으로서 보편적인 Unix Domain Socket 을 활용한 익스플로잇이 존재[1, 2]하였으나, 최근 구글 및 여러 개발자들의 노력을 통해 SELinux Enforce 상태에서 System 및 OEM 서명을 제외한 프로세스의 경우 Socket 기반 IPC를 차단하는 등의 DAC 기반 Sandbox 프로세스 격리를 구현[3]하여 이러한 권한 상승 익스플로잇을 미연에 방지하고자 하였다.

하지만, 그러한 노력에도 불구하고, 또 다른 Android의 IPC 파이프라인인 Binder 의 경우 Android 프레임워크와 HAL에 의해 큰 의존성을 가지고 있었고[4], 따라서 Domain Socket IPC 와 같은 강력한 제한을 적용하기는 어려웠기 때문에, 이와 같은 연유로 인해 Binder를 기반으로 하는 다른 IPC 기법인 BroadcastReceiver 와 ContentResolver에 대한 취약점이 상당수 존재하였다.[5, 6] 따라서, 본 연구에서는 Binder 기반의 IPC 기법인 BroadcastReceiver 와 ContentResolver 을 사용하여 현재까지도 존재하는 이들 IPC 체계의 취약점을 증명하고, 민감 API 권한에 존재하는 제약을 통합 및 회피 하기 위한 솔루션을 개발 및 제안하고자 한다.

2 Design

먼저 IPC 데이터 흐름을 설계하기 앞서서, API 의 요구 사항을 결정하기 위해 어떤 API의 기능을 구현할

것인지에 대해 논의하였다. 구글의 민감 API에 대한 문서에 따르면, 민감 API는 다음과 같은 카테고리로 정의되어 있다:

- SMS & Phone Call State
- Geo Location
- File I/O
- Package Query & Management
- Accessibility
- VPN Service
- Exact Alarm Creation
- Full Screen

이 중 Accessibility, VPN Service, Exact Alarm Creation, Full Screen 의 경우 Intent 및 Context 와 밀접하게 연관되어 대상 애플리케이션과의 주기적인 상호 작용이 필요하지만, 본 프레임워크에서 사용하는 IPC 기술은 단편적인 정보 교환을 주 목적에 두기 때문에 본 프레임워크에서의 구현은 적합하지 못하다. 또한, 이러한 기능에 대한 구현은 Binder 기반 IPC보다는 ClassLoader가 효과적이지만[7], BroadcastReceiver 의 근본적인 제약으로 인해 적합하지 못하여 API 구현 대상에서 제외하였으며, 이에 대한 구체적인 이유는 챕터 4–2 에서 자세히 다루도록 한다.

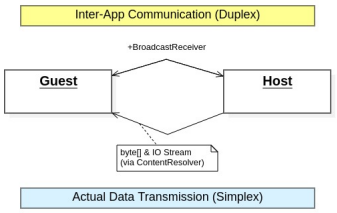

Figure 1: 호스트-게스트 간 통신 개관

이 프레임워크의 전체적인 토폴로지에 관해서는, 먼저 역할을 두가지로 나눈다. 하나는 시스템으로부터 직접적으로 민감 API에 접근할 수 있는 권한을 가진 호스트(Host) 이며, 나머지 하나는 호스트로부터 본 프레임워크를 사용하여 민감 API의 정보를 받는 게스트(Guest) 역할로 나눌 수 있다.(Figure 1) 이때, 하나의 앱은 호스트임과 동시에 게스트 역할도 겸할 수 있으며, 그 역도 성립 가능하다. 또한, 호스트 또는 게스트는 여러개의 게스트/호스트와 상호 작용 가능하도록 한다. (Figure 2)

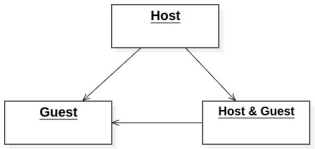

Figure 2: 여러 개의 노드 간 통신 개관

2-1 Protocol Design

본 프레임워크에서는, Intent 를 매게로 하여 데이터를 전송하는 BroadcastReceiver (이하 Receiver)를 앱 간 Handshake 및 매타데이터 송수신 용도로 전용하고, 실질적인 데이터의 교환은 ContentResolver 의 IOStream 전송을 통해 전송하도록 하였다.(Figure 3) 이로서 대규모의 데이터를 Binder를 통해 전송하면 발생하는 문제인 1MB Transaction Buffer 크기 제한 문제를 해결하고,[8], 데이터 수송신 시 BroadcastReceiver에서의 작업 시간 소요로 인한 백그라운드 Thread ANR 문제를 조기에 예방하기 위해서이며, 이때 사용되는 데이터의 수송신 요청 흐름은 다음과 같다:

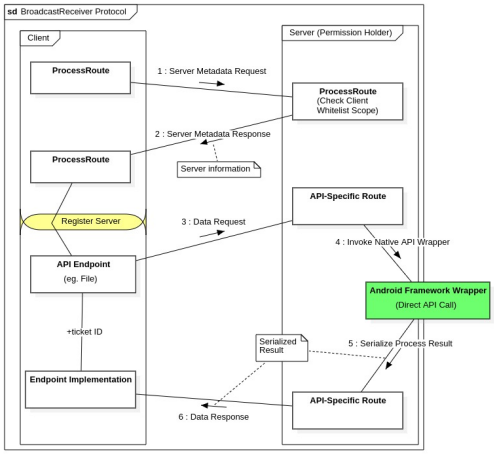

Figure 3: BroadcastReceiver IPC 프로토콜 개관

첫 번째로, 먼저 클라이언트는 프레임워크가 시작되면 AmusePermit 을 사용하는 모든 에플리케이션에 공통적으로 상수로서 선언되어 있는 Intent Action 이름을 토대로 사용 가능한 피어를 탐색한다. 한번 서버를 탐색하게 되면, 클라이언트는 Receiver를 통해 피어의 메타데이터를 요청하고,(과정 1) 피어는 이에 피어를 패키지 이름을 토대로 검증한 후 즉각 자신의 메타데이터 정보를 반환한다.(과정 2) 이 과정은 본격적으로 데이터가 송수신 되기 전 피어 간 핸드셰이크를 수행하므로서 앞으로의 통신에 필요한 기본적인 정보를 가져오기 위한 것이다. 이때 이 과정 중 피어 간 전달되는 메타데이터에는 피어의 패키지 명, 피어의 서버/클라이언트 유무, 피어가 서버일 경우 서버가 제공하는 API의 종류, 해당 서버가 피어를 햐용하는지 등에 대한 정보가 담겨 있으며, 이 핸드셰이크 과정이 종료되면 클라이언트는 이 정보를 AmusePermit 에 등록함으로서 해당 피어를 서버로 사용할 것임을 알림으로서 본격적인 데이터를 송수신할 준비를 마친다.

이후 클라이언트가 엔드포인트 API를 호출해 데이터를 요청하면, 클라이언트는 호출한 API 의 정보 및 매개변수를 Receiver를 통해 요청을 서버에게 전달하며, (과정 3) 서버는 이 정보를 토대로 Android Native API 의 Wrapper 클래스를 토대로 메서드를 실행하고, (과정 4) 이때 반환되는 실행 결과의 반환값을 직렬화하여 다시 클라이언트에게 반환한다 (과정 5 6) 이후 클라이언트는 받은 데이터를 역직렬화햐여 객체의 형태로 변환 후 최종적으로 엔드포인트 API 의 비동기 리스너를 통해 결과값을 애플리케이션으로 전달하게 된다. 이때 개별적인 API의 요청들을 구분하고자, 각 API의 호출은 티켓의 형태로 고윳값을 가지게 되는데, 이 고윳값들은 전체 데이터 처리 파이프라인에서 피어 간 Receiver 송/수신에 반드시 포함되어 엔드포인트 API로의 결과값 전송 시 비동기 구현으로 인한 리스너 호출에 사용되어진다.

또한 만약 전달하고자 하는 데이터가 파일 내용과 같이 큰 경우, 앞서 언급했듯이 Receiver 가 아닌 ContentResolver 를 통해 IO Stream의 형태로 데이터의 전달을 진행하게 되는데,(Figure 4) 먼저 API가 Stream 이 필요한 API를 호출하게 되면, 위의 경우와 마찬가지로 클라이언트는 호출한 API 의 정보 및 매개변수를 Receiver를 통해 요청을 서버에게 전달하며, (과정 1) 서버는 Android Native API 를 거쳐 필요한 IO Stream 의 객체를 생성하게 된다.(과정 2)

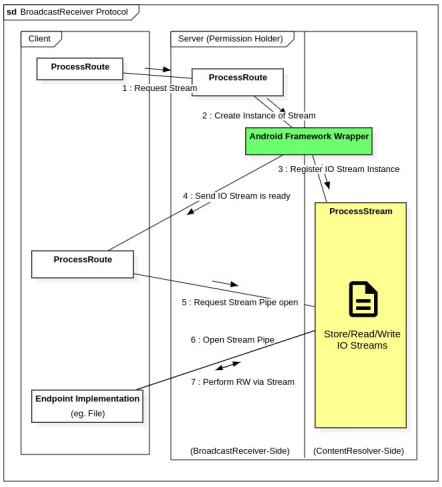

Figure 4: ContentResolver 통신 개관

이후 IO Stream 의 객체를 ContentResolver에 등록하고, (과정 3) 데이터 전송을 위한 Stream 이 열렸다는 것을 클라이언트에게 알리면, (과정 4) 클라이언트는 Receiver 가 아닌 Uri 와 Provider를 통해 ContentResolver 에 접근하게 되고, (과정 5) 이후 서버에서는 ContentResolver 에서 Stream의 Pipe를 열어 Input 또는 Output 한쪽의 Stream 중 클라이언트가 원하는 것을 File Descriptor 을 통해 전달하게 됨과 동시에 클라이언트의 read 혹은 write 처리를 위한 스레드를 시작한다.(과정 6) 이후 클라이언트가 이 Stream 에 대한 read 혹은 write 를 수행하면 서버는 열어 놓은 스레드를 통해 데이터를 전달하게 되며,(과정 7) 데이터 전송이 끝나면 스레드와 Pipe 를 닫고 Stream 객체를 삭제하여 작업을 마무리한다.

2-2 Serialization Implementation

본 프레임워크를 사용하는 애플리케이션에게 인터페이스를 제공하고, 처러된 결과값을 반환하는 클래스

의 일종인 Endpoint API에 본 프레임워크에서 필요로 하는 모든 메서드, 필드들을 한번에 구현하게 될 경우 하나의 클래스에 모든 데이터가 있기 때문에 직렬화에는 효과적일지 모르나, 이들 클래스는 용도에 따라 필드와 메서드들을 구분하기 어려워짐으로서 추후 개발의 유지보수에 있어 애로사항이 될 수 있다. 따라서, 본 프레임워크에서는 개발의 효율성을 위해 용도에 따라 Endpoint 클래스를 세가지로 분류하여 구현하였다.(Figure 5)

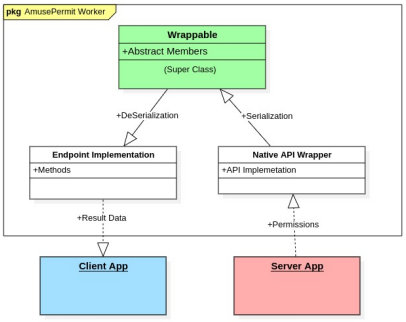

Figure 5: 직렬화 과정 모델

먼저, Wrappable 클래스의 경우 API 형식마다 고유한 DTO Model 클래스로서, Endpoint 구현 클래스에서 getter/setter 메서드들을 통해 애플리케이션으로 전달할 필드들을 선언한 클래스이며, 이 클래스에서 직렬화 ID 를 두어 다른 API 형식들과 구분한다. 또한 Endpoint 구현 클래스의 경우 Wrappable을 상속하여 클라이언트 애플리케이션에게 최종적으로 인터페이스와 데이터를 제공하는 클래스이다. 마지막으로 Native Wrapper 의 경우 마찬가지로 Wrappable을 상속한 클래스로, 민감한 Android Native API 로부터 직접적으로 상호작용하는 DAO 클래스이며, 이는 서버 역할의 애플리케이션에서 프레임워크 내부적으로 사용하는 클래스로, 따라서 Endpoint 구현 클래스와 달리 서버 애플리케이션에게 노출되지 않을 뿐만 아니라 일반적으로 민감한 API와 직접적으로 상호작용 하기 때문에 Manifest 에 권한의 선언이 필요하다.

처리 과정의 경우 일반적으로 Native Wrapper 클래스에서 시작하여 Wrapper에서 필요한 데이터를 확보한 후 직렬화를 위해 Wrappable 클래스로 변환하여 서버에서 클라이언트로 전달하며, 이후 클라이언트는 Endpoint 클래스의 형태로 역직렬화 및 형변환을 통해 클라이언트 애플리케이션에게 제공하게 된다. 또한 역으로 Endpoint 에서 출발하여 처리될 수도 있는데, 이는 사용자가 Endpoint애서 필드 데이터를 변경하여 Native API에 상호 작용하고자 하는 경우이므로 이 경우에는 클라이언트에서 Endpoint 클래스를 그대로 직렬화하여 서버로 전달, 이를 토대로 Native Wrapper 클래스를 생성하여 Native API와 상호작용을 하게 된다.

2-3 Static analysis interference

일반적으로 민감 권한 API 의 사용을 탐지하는 솔루션의 경우, Android Manifest 에서 선언된 권한을 확인하거나, APK 를 디컴파일 하여 Smali 바이트코드를 정적 분석하여 특정 API가 사용되었는지 확인하는 방식이 주로 사용된다.[9, 10] 이때, 본 프레임워크에서 Native Wrapper의 구현을 위하여 아무런 조치 없이 Native API를 사용시 클라이언트 애플리케이션이 실제로 Native API를 직접적으로 사용하지 않고, 본 프레임워크를 통해 간접적으로 사용하더라도 이러한 정적 분석 기법을 사용하는 경우 클라이언트 애플리케이션이 실질적으로 민감한 API를 사용하는 것으로 판단 될 가능성이 있다. 따라서, 본 프레임워크에서는 이러한 문제점을 보안하기 위해, Native Wrapper을 별도의 독립적인 모듈로 분리시키고 서버 애플리케이션에만 이를 포함시킴으로서 이러한 문제점을 해결하고자 하였다.

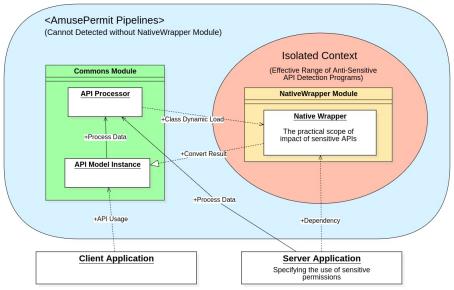

Figure 6: Native Wrapper 격리 개관

직렬화 과정에서도 언급했듯이, API Processor 에서 클라이언트의 요청으로 Native Wrapper 가 호출

될 때, API Processor 는 Native Wrapper를 정적으로 호출하지 않고, 애플리케이션이 Native Wrapper 모듈을 탑재했는지에 대한 여부를 판별하여 Class Name 을 토대로 Instance를 생성하여 동적으로 호출하게 된다. 이때, 클라이언트 애플리케이션 에서는 결과값을 얻기 위해 Native Wrapper의 인스턴스를 역직렬화 할 경우 이에 대한 모듈이 탑재되어 있지 않아 클래스의 Link 가 실패할 것 임으로, 서버 애플리케이션은 Native Wrapper 을 클라이언트로 송신하기 전 Wrappeable 형태의 DTO Model 클래스로 변환하여 이를 방지한다.

3 Result Analysis

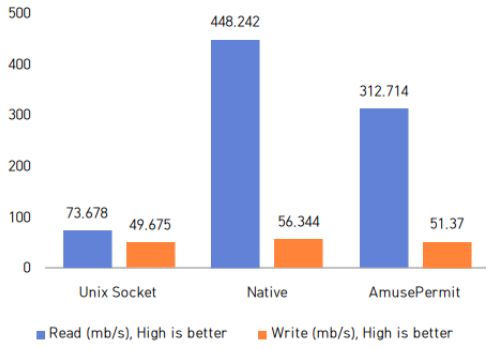

Figure 7: File I/O 구현 방법 별 대역폭 비교

본 프레임워크의 성능을 측정하기 위해 정량적인 데이터를 도출하기 가장 용이한 방법인 파일을 이용한 벤치마킹을 실시하였다. 벤치마킹에서는 읽기 및 쓰기 시의 대역폭과 레이턴시를 각각 측정하였고, 테스트 기기로는 64GiB eMMC 5.1, 4GiB LPDDRX4, Linux 4.19 및 Android 13을 탑재하고 Ext4 FS를 사용하는 Xiaomi Note 7 기기를 사용하였으며, 측정 기준으로는 레이턴시의 경우 File 객체를 생성하여 IO Stream 의 파일 Pipe가 열릴 때까지를 기준으로 하였으나 벤치마킹을 하기 위한 폴더나 파일을 생성하는 등의 준비하는 시간은 측정에서 제외하였고, 대역폭의 경우에는 vnemes가 작성한 AndroidBenchmark 의 소스코드를 참조하여 1 GiB 의 파일을 4KiB의 고정 버퍼를 통해 쓴 다음 메모리를 비우고 쓴 파일을 다시 읽는 시간을 각각 측정하는 방식의 벤치마킹을 10번 반복하여 평균을 계산해 결과를 도출하였으며, 측정 당시의 단위는 마이크로초 단위로 데이터를 수집하였으나 본 논문에서는 편의상 밀리초 단위로 변환하여 나타내었다.

대역폭의 경우, 기존의 Unix Socket, ContentProvider 기반의 본 논문에서 제시된 AmusePermit, 그리고 대조군으로 Native 구현을 두고 읽기 및 쓰기 대역폭을 비교하였다.(Figure 7) 쓰기의 경우 세 구현 방식 모두에서 50MB/s 전후의 대역폭을 보이며 큰 차이를 보이지 않았으나, 읽기의 경우 Native 구현 대비 AmusePermit 이 69.76%, Unix Socket 이 16.43% 의 성능을 보여 열세인 것으로 보이나, Unix Socket과 비교하여 AmusePermit 이 53.32%p 가량 우세하므로 IPC 통신 중 대규모 데이터 전송에 있어서 AmusePermit 이 Unix Socker 보다 적합하다고 볼 수 있다.

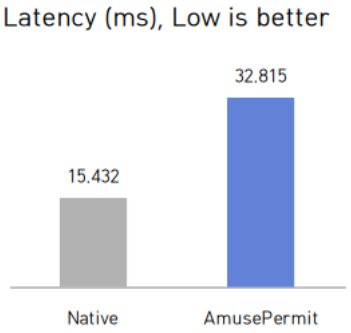

Figure 8: Native 구현 대비 레이턴시 비교

지연시간의 경우, 스토리지 상에서의 파일 처리 시 발생하는 대부분의 지연시간은 파일의 읽기 및 쓰기를 준비하기 위한 벡엔드와 프론트엔드 간 이중(Duplex) 통신에 기인하므로, 따라서 AmusePermit 에서 벡엔드와 프론트엔드 간 파일의 메타데이터 전송에 소요되는 시간을 측정하기 위해 Native 와 AmusePermit의 퍼일 IO Stream 지연 시간을 측정하였으며, Unix Socket의 경우 동일한 측정 방식을 만족하기 위해서는 별도의 벡엔드 구현이 필요하지만, 구현시 BroadcastReceiver와 같은 AmusePermit과 사실상 동일한 구현 방식이 사용되므로 Unix Socket은 측정에서 제외하였다. 측정 결과, Native 구현의 지연시간 측정값의 평균은 약 15ms 로, eMMC 의

일반적인 레이턴시가 16ms 이내인것을 감안할때[11] 측정값의 신뢰도는 높은 것으로 보이며, AmusePermit의 경우 32ms 로 약 47% 가량 높은 것으로 나타났으며, 이를 통해 순수 AmusePermit의 통신에 사용되는 시간은 AmusePermit의 지연시간에서 Native 구현의 지연시간을 뺀 약 17ms 로 추측해 볼 수 있다. 이는 사용자에게 있어 대기 시간이 주로 약 100ms 후에 눈에 띄게 나타나는 것을 감안하여 볼때,[12] 이러한 AmusePermit의 적은 지연시간은 UX(User-Experience) 측면에서 이점이 있다고 볼 수 있다.

4 Discussion

4-1 Expected Usage

일반적으로 이 프레임워크는 하나의 애플리케이션에서 단독으로 작동하지 아니하며, 복수의 애플리케이션이 서로 상호작용하며 작동하도록 설계되었다. 따라서, 본 섹션에서는 이러한 프레임워크를 주로 사용될 것으로 예상되는 애플리케이션 간 상호작용에 관한 세가지 모델을 제시한다.

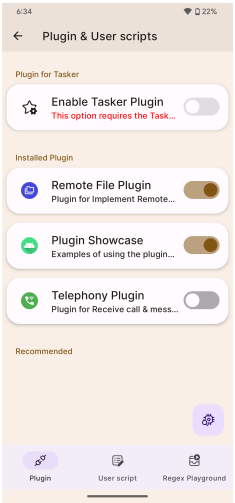

Figure 9: 상용 애플리케이션의 플러그인 모델 적용 예

Master-Slave 모델이라고도 불리는 플러그인 모델(Figure 10)은 의 사례와 같이 소수의 Master 애플리케이션이 사용자에게 비즈니스 코드를 비롯한 주 기능을 제공하고, 다수의 Slave 애플리케이션은 이 프레임워크를 사용하여 민감 권한 API 관련 데이터만을 제공하는 역할만을 수행하는 형태를 의미한다.

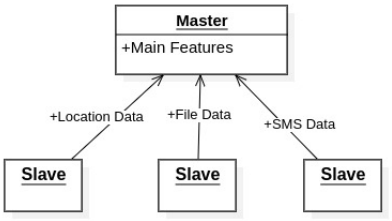

Figure 10: 플러그인 모델

이 모델은 주로 Master 애플리케이션이 플러그인의 형태를 띄는 Slave 애플리케이션을 관리하는 형태로 구현된다. 예를 들어, Figure 9 사례의 경우 Master 애플리케이션의 권한 부여의 집중을 방지하고자 각각 File 플러그인과 전화/메시지 플러그인의 형태로 분리하여 사용자가 원하는 권한과 기능만을 선택할 수 있도록 해 보다 유연한 사용자 환경을 구현하였다.

Figure 11: Pure-P2P 식 모델

Pure P2P 모델은 애플리케이션 간의 뚜렷한 상하관계 없이, 서로가 가지고 있는 권한을 이용하여 피어간 민감 API 데이터를 공유하는 형태를 나타낸다. 이들 각각의 애플리케이션은 서로의 주 기능을 제공하면서 필요시 상호 보완하는 관계로 구현되며, 이는 주로 다중의 애플리케이션 개발을 통해 생태계 구축을 지향하는 개발자에게 사용되어질 수 있는 모델이다.

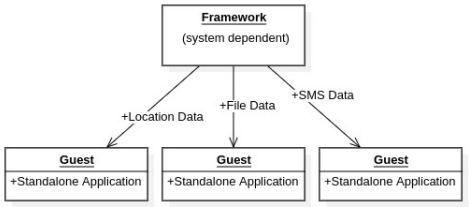

Figure 12: 프레임워크 식 모델

프레임워크 식 모델은 소수의 민감 권한 API 관련 데이터만을 제공하는 역할만을 수행하는 애플리케이션이 다수의 주 기능을 제공하는 애플리케이션에게 데이터를 공유하는 모델로서, 주로 다수의 애플리케이션이 동일한 민감 API 데이터에 접근하고자 할 때 사용되어질수 있으며, 이 모델은 주로 시스템 관련 개발자가 운영체제 및 임베디드 프레임워크 개발과 같은 분야에 사용될 것으로 예상된다.

4-2 Limitations

본 프레임워크를 Classloader 및 Unix Socket Domain 과 같은 직접적으로 Class를 주입하여 원본 애플리케이션의 프로세스를 벗어나 권한을 가지고 있는 애플리케이션의 프로세스에서 실행되는 기법들과 비교하여 볼때, 커널이 Class 가 실행되는 PID가 권한을 가지고 있는 앱으로서 인식하기 때문에 별다른 수정 없이 직접적으로 Native API를 사용[13]할 수 있는 타 기법들과 달리, 직렬화 데이터만을 주고 받을 수 있는 본 프레임워크의 구조적 한계상 부모 프로세스와 와 긴밀하게 상호작용하는 Context 기반의 API들 (Activity, BroadcastReceiver 등) 의 사용이 불가능하다는 제약 사항이 존재하며, 이는 Manifest 에 정적으로 선언해야 사용할 수 있는 API 들 (Service, Boot-Complete Receiver 등) 에도 마찬가지로 적용된다. 이를 해결하기 위해서는 PendingIntent 를 IPC를 통해 전송하는 기법이 존재하나,[14], 실제로 이를 AmusePermit에 적용 가능한지에 대해서는 추가적인 연구가 필요해 보인다. 또한, Native API 를 사용하기 위해서는 위와 같은 이유로 인해 Wrapper 의 구현이 필수적인데, 이는 Native API 를 사용하기 위해 기존 코드를 수정하거나 새로 작성함이 불가피하다는 단점을 가지며, 새로운 API 또는 기존 API 변경사항 반영에 대한 필요가 발생할 경우 Native 구현 대비 개발에 대한 AmusePermit 단의 추가적인 자원이 투입되어야 한다는 단점 또한 존재한다.

5 Related Works

Android의 민감한 API의 누출 문제는 이전에 많은 문헌에서 연구되었다. SAUSAGE는 Android의 Unix 도메인 소켓 사용량에 대한 보안을 분석함에 있어[15] 액세스 제어 문제가 있어 신뢰할 수 없는 앱이 하드웨어 제조업체 또는 공급업체 사용자 정의에서 도입한 Unix 도메인 소켓을 통해 높은 권한을 가진 데몬과 통신할 수 있음을 입증하였으며, IacDroid 는 이와 관련하여 IAC(Inter-App Communication) 에 있어 Binder IPC 메커니즘과 시스템 서비스를 확장하여 여러 애플리케이션 간에 컨텍스트 기반 구성 요소 호출 체인을 구성하여[16] 콜 체인을 활용하여 권한 시스템이 확장되어 IAC 기능 누출을 감지하고 방지하는 솔루션을 제안하였다. 또한, ComDroid는[17] Android의 Dex 를 정적으로 역분석함으로서 BroadcastReceiver 를 비롯한 Intent 기반의 IPC 를 분석하여 멀웨어를 탐지하는 솔루션을 제안하였다.

6 Conclusion

본 논문에서는, BroadcastReceiver와 ContentResolver 의 이종 IPC 통신을 활용하여 서버-클라이언트 애플리케이션 간 데이터를 공유할 수 있는 프로토콜을 설계하고 데이터 Pipe 터널을 구현함으로 앱이 가지고 있는 권한을 다른 앱에게도 공유하면서 최대한 Native API를 쓰는 것과 비슷한 개발 경험을 제공하고자 하는 API 엔드포인트 설계에 집중하였으며, 기존의 SeLinux 제한을 충족함과 동시에 Android 의 민감한 권한을 사용하는 API에 대한 제약을 우회하여 앱 별로 개별적으로 권한을 부여해야 하였던 불편함을 해소하는데 성공하였다. 또한, 구축한 프레임워크를 토대로 File IO 파이프라인을 구현하여 기존의 Native 구현 및 Unix Socket 과 대역폭과 레이턴시를 측정하였으며, 벤치마킹 결과 Native 구현대비 성능적인 측면이 열세를 가지지만 Unix Socket 대비 우위를 점하는 것으로 나타났다.

한편으로는 Android의 SELinux 도입 이후 일반적인 Third-Party Application 간 Unix Domain Socket IPC 의 사용이 차단됨에 따라 더이상 사용할 수 없

었던 IPC 취약점을 이용한 민감 API에 대한 우회 기법이 여전히 존재한다는 측면을 본 프레임워크를 반례로서 PoC(Proof-of-Concept) Exploit 으로 제시함으로서 Android 생태계의 보안에 대해 고찰하는 계기를 제공하였다.

7 Appendix

이 연구에 사용된 모든 소스코드는 GPL 3.0 라이선스에 따라 다음 저장소에 공개되어 있음:
https://github.com/choiman1559/AmusePermit

References

[1] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z Morley Mao. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 80–91, 2016.

[2] Xu Jiang, Dejun Mu, and Huixiang Zhang. Unix domain sockets applied in android malware should not be ignored. *Information*, 9(3):54, 2018.

[3] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. Analysis of seandroid policies: Combining mac and dac in android. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC '17, page 553–565, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Nitay Artenstein and Idan Revivo. Man in the binder: He who controls ipc, controls the droid. In *Eur. Black-Hat Conf*, 2014.

[5] Fadi Mohsen, Halil Bisgin, Zachary Scott, and Kyle Strait. Detecting android malwares by mining statically registered broadcast receivers. In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 67–76, 2017.

[6] Hossain Shahriar and Hisham M. Haddad. Content provider leakage vulnerability detection in android applications. In *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, page 359–366, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[8] Google developer reference documentation, about transactiontoolargeexception (https://developer.android.com/reference/android/os/transactiontoolargeexception).

[9] Chunlei Zhao, Wenbai Zheng, Liangyi Gong, Mengzhe Zhang, and Chundong Wang. Quick and accurate android malware detection based on sensitive apis. In *2018 IEEE international conference on smart internet of things (SmartIoT)*, pages 143–148. IEEE, 2018.

[10] Meng Shanshan, Yang Xiaohui, Song Yubo, Zhu Kelong, and Chen Fei. Android's sensitive data leakage detection based on api monitoring. 2014.

[11] Deng Zhou, Wen Pan, Wei Wang, and Tao Xie. I/o characteristics of smartphone applications and their implications for emmc design. In *2015 IEEE International Symposium on Workload Characterization*, pages 12–21. IEEE, 2015.

[12] Nancy P. Ambritta, Damini, and Harshad P. Bhandwaldar. A survey on real-time application latency based on android os. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pages 1–4, 2018.

[13] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. Dydroid: Measuring dynamic code loading and its security implications in android applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–426, 2017.

[14] Chennan Zhang, Shuang Li, Wenrui Diao, and Shanqing Guo. Pitracker: Detecting android pendingintent vulnerabilities through intent flow analysis. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '22, page 20–25, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Mounir Elgharabawy, Blas Kojusner, Mohammad Mannan, Kevin R. B. Butler, Byron Williams, and Amr Youssef. Sausage: Security analysis of unix domain socket usage in android. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroSP)*, pages 572–586, 2022.

[16] Daojuan Zhang, Rui Wang, Zimin Lin, Dianjie Guo, and Xiaochun Cao. Iacdroid: Preventing inter-app communication capability leaks in android. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 443–449, 2016.

[17] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 239–252, New York, NY, USA, 2011. Association for Computing Machinery.