Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.1, Feb. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.1.205

안드로이드 앱 캐시 변조 공격의 설계 및 구현*

홍 석,[1†] 김 동 욱,[1] 김 형 식[2‡]
[1]삼성전자 Samsung Research, [2]성균관대학교

A Practical Design and Implementation of Android App Cache Manipulation Attacks*

Seok Hong,[1†] Dong-uk Kim,[1] Hyoungshick Kim[2‡]
[1]Samsung Research, Samsung Electronics, [2]Sungkyunkwan University

요 약

안드로이드는 앱 캐시 파일을 사용하여 앱 실행 성능을 향상시키고 있지만, 이런 최적화 기술은 검증 과정의 보안 문제를 야기할 수 있다. 본 논문에서는 개인 정보를 유출시키거나 악성 행위를 수행하도록 악용하기 위해 공격 대상 앱의 앱 캐시 파일을 변조하는 "안드로이드 앱 캐시 변조 공격"에 대한 실용적인 디자인을 제시한다. 공격 설계의 타당성을 입증하기 위해 공격 도구를 구현하고 실제 안드로이드 앱을 대상으로 실험을 수행했다. 실험 결과에 따르면 29개 앱 중 25개 앱(86.2 %)이 해당 공격에 취약한 것으로 확인되었다. 안드로이드 프레임워크는 체크섬 기반의 무결성 검사를 통해 앱 캐시 파일을 보호하고 있으나, 앱 캐시 파일에 저장된 체크섬 값을 변조함으로써 효과적으로 해당 보호 방법을 우회할 수 있음을 확인했다. 안드로이드 앱 캐시 변조 공격에 대응하기 위한 2가지 가능한 방어 방법으로 (1) 앱 캐시 파일의 무결성 검사 방법과 (2) 디컴파일 방지 기술을 제안한다.

ABSTRACT

Android uses app cache files to improve app execution performance. However, this optimization technique may raise security issues that need to be examined. In this paper, we present a practical design of "Android app cache manipulation attack" to intentionally modify the cache files of a target app, which can be misused for stealing personal information and performing malicious activities on target apps. Even though the Android framework uses a checksum-based integrity check to protect app cache files, we found that attackers can effectively bypass such checks via the modification of checksum of the target cache files. To demonstrate the feasibility of our attack design, we implemented an attack tool, and performed experiments with real-world Android apps. The experiment results show that 25 apps (86.2%) out of 29 are vulnerable to our attacks. To mitigate app cache manipulation attacks, we suggest two possible defense mechanisms: (1) checking the integrity of app cache files; and (2) applying anti-decompilation techniques.

Keywords: ART, app cache, app integrity attack

I. 서 론

하드웨어 및 소프트웨어 기술이 발전됨에 따라 뛰어난 휴대성을 지닌 스마트폰은 일상과 업무를 넘나드는 필수품이 되었다. 이에 따라, 기존의 많은 서비스가 스마트폰 애플리케이션(이하 앱)을 통해 제공되고 있으며, 지속해서 증가하는 추세이다.

이러한 기술 발전의 이면에는, 앱을 위/변조하여

Received(12. 17. 2018), Modified(01. 24. 2018), Accepted(01. 28. 2019)
* 본 연구는 삼성전자 Samsung Research 지원 및 관리로 수행하였습니다.
† 주저자, seok85.hong@samsung.com
‡ 교신저자, hyoung@skku.edu(Corresponding author)

악성 코드를 삽입하거나 지급되지 않은 앱의 기능을 임의로 해제하는 것과 같은, 금전적인 이득을 위한 공격이 증가하는 위협도 발생하고 있다. 특히, 안드로이드는 전 세계 모바일 운영체제 중 약 75.66%의 점유율을 가진 가장 널리 사용되는 운영체제 중 하나로서 [1], 이러한 위협에 가장 많이 노출되어 있다. 안드로이드 앱에서의 위/변조 공격은 매우 흔하며, 루트 권한을 임의로 획득하여 시스템을 변조하고 비공식 앱스토어를 통한 앱을 설치, 사용하는 등[2]의 특수한 목적을 가진 일부 사용자들로 인해 안드로이드 앱에 대한 보안 위협은 더욱 증가하고 있다.

안드로이드에서 구동되는 앱은 APK (Android App Package) 파일 형식으로 배포되는데, 해당 파일 형식은 ZIP 파일 형식의 확장 형태이기 때문에 쉽게 APK 파일의 내부 데이터를 확인, 추출할 수 있으며, 달빅 바이트코드로 구성되어 있는 앱 소스코드는 Smali/Apktool과 같은 오픈 소스 분석 도구를 사용하여 앱에 대한 소스코드 디컴파일, 분석, 변조 및 삽입과 같은 공격을 쉽게 수행할 수 있다[3,4]. 이에 따라 안드로이드 앱 보안에 대한 연구는 APK (Android application package)를 보호하는 연구가 진행되어 왔으나[5,6,7,8], 안드로이드 5.0부터 공식 도입된, 새로운 앱 실행 환경인 Android Runtime(이하 ART)이 등장한 이후로 많은 어려움에 봉착하게 되었다.

안드로이드는 ART에서 앱 실행 시, 더 이상 APK 파일을 사용하지 않는 대신, 앱 설치 시 APK 파일로부터 생성한 앱 캐시 파일을 직접 로딩하기 시작하였다. ART는 앱의 실행 성능 최적화를 위해, 디바이스의 아키텍처 및 실행 상황을 고려하여 다양한 방식의 최적화 기법을 사용하여 앱 캐시 파일을 생성하기 때문에 가변성이 매우 높다. 이러한 환경 아래서, Sabanal[9]은 앱 캐시 파일의 조작을 통해 안드로이드 프레임워크 및 앱의 동작을 변경하여 사용자 레벨 루트킷을 구현하는 연구를 발표하였으나, 사용자가 알아차릴 수 없는 안드로이드 시스템의 구조와 앱 캐시 파일 변조 공격에 대응할 수 있는 보안 솔루션이 부재한 상태이다.

본 논문에서는 최신 ART 앱 캐시 파일에 관련된 파일 포맷과 구동 방식에 대한 시스템 분석을 수행하여 그 결과를 정리하고 Sabanal이 제안한 앱 캐시 변조 공격에 대한 실용적인 공격 설계와 해당 공격에 대한 유효성 실험 결과를 보였다, 또한 기존 공격 모델에서 실패한 일부 앱들에 대해 공격 실패 원인을 분석하고 리소스 보호 솔루션 우회, 앱 캐시 파일에 대한 직접 변조와 같은 추가 공격 모델을 설계 및 제안하였다. 마지막으로, 앱 캐시 무결성 검사 방법과 디컴파일 방지 기술에 대한 2가지 방어 기법 아이디어를 제안한다.

II장에서는 안드로이드 앱 실행 환경인 Android Runtime(ART)에 대한 시스템 분석한 결과를 설명하고 III장에서는 안드로이드 앱 캐시 변조 공격에 대한 설계 및 효용성 실험 결과를 보이고, 마지막으로 IV장에서 해당 공격에 대한 방어 기법을 제안하고 V장에서 결론을 맺도록 한다.

II. 배 경

2.1 ART (Android Runtime)

안드로이드는 앱의 실행 성능 개선을 위한 노력의 산물로 안드로이드 4.4 Kitkat에서 ART를 처음 공개하였으며 이후 안드로이드 5.0 Lollipop부터 기본 앱 실행 환경으로 채택되어 기존의 DVM(Dalvik Virtual Machine)을 대체하였다.

이후 꾸준한 발전으로 안드로이드 8.0 Oreo 기준으로 DVM, JIT (Just in Time Compiler), AOT (Ahead of Compiler)와 같은 다양한 최적화 기법을 사용하는 하이브리드 형태를 가지고 있다[10,11,12,13,14].

2.1.1 ProfileSaver 스레드

ProfileSaver 스레드는 ART에서 도입된 백그라운드 데몬의 하나로, ART에서 앱을 실행하는 도중 분석된 클래스와 메소드간의 정보를 수집하고 JIT에 의해 컴파일 되거나 인터프리터에 의해 해석된 모든 메소드에 대해서 특정 임계값을 초과할 만큼 자주 호출되는 메소드들을 /data/misc/profiles에 이들의 색인 정보를 저장하는 역할을 가지고 있다. 이 파일을 프로파일이라 하며, AOT 컴파일러는 이 파일에 기록된 메소드들을 컴파일하여 네이티브 코드(Native code)로 변환한다.

2.1.2 base.art 파일

안드로이드 ART에서 Dex를 포함한 앱 캐시 파일을 로딩하고 실행할 때 클래스들과 메소드에 대한

링킹과정을 수행한다. 이런 현상은 특히 앱 실행 상황에 따라 그 링킹 정보가 바뀔 수 있는 가상 메소드를 처리할 때 두드러진다. 앱 이미지 파일(base.art)은 매번 앱을 실행할 때마다 이런 클래스와 메소드들의 링킹 과정을 재수행하는 비효율을 줄이기 위해 생성된 캐시 파일의 일종으로, 런타임에 기록된 클래스 룩업테이블을 저장하고 있으며 앱 캐시 파일을 로딩 하는 과정 중에 base.art 파일이 존재할 경우 해당 파일을 열어 미리 계산된 링킹 정보를 반영함으로써 효율을 향상시키고 있다.

2.1.3 AOT 컴파일 과정과 컴파일러 필터

AOT 컴파일러는 ART를 사용하는 안드로이드 디바이스의 특정 조건 (예: 디바이스가 충전 중이며 유휴 상태일 때 등)이 만족되면 실행되는 데몬의 일종을 말하며, AOT 컴파일 데몬은 ProfileSave 스레드가 저장한 프로필 파일을 읽고 기록된 앱들의 메소드들을 네이티브 코드로 최종 컴파일하여 앱의 실행 속도를 향상시키는 역할을 가지고 있다.

컴파일러 필터는 AOT 컴파일러가 동작할 때 앱 캐시 파일의 최적화를 제어하기 위해 소개된 개념으로 디바이스를 제조한 제조사의 최적화 정책에 따라 달라질 수 있다. 기본 아이디어는 앱이나 프레임워크 라이브러리를 디바이스의 성능 및 하드웨어 조건에 맞춰 다양한 모드로 컴파일하여 결과적으로 CPU 사용량, 배터리, 디스크 사용량, 설치 시간, 앱 실행 시간 등을 적절하게 최적화하는데 있다.

안드로이드는 6.0을 시작으로 다양한 시도를 통해 최적화 방식을 제안하고 있다. 맨 처음 AOT 컴파일러가 소개된 안드로이드 6.0에서는 앱 실행 시간의 최적화를 가능하게 했지만, 앱을 설치하는 데 너무 많은 시간과 자주 호출되지 않는 메소드 마저 네이티브 코드로 컴파일하기 때문에 앱 캐시 파일을 위한 많은 공간을 소비했다. 이에 따라 다양한 컴파일 옵션을 제공함으로써 디바이스의 성능과 상황에 적절한 앱 실행 시간을 가지면서도, CPU 사용량, 배터리 사용량, 저장장치 사용량을 조절할 수 있게 되었다. 안드로이드 7.0에는 총 12개의 컴파일러 필터가 존재하며, 안드로이드 8.0에서는 그보다 적은 4개의 필터가 공식적으로 지원되었다.

2.1.4 Dex-to-Dex 최적화

AOT 컴파일러와 컴파일러 필터를 사용하여 네이티브 코드를 컴파일하는 최적화와 별개로 달빅 바이트코드 자체의 최적화 역시 시도되었다. 달빅 가상머신이 설치되어있다면 어디에서든지 실행 가능해야 하는 달빅 바이트코드의 특성으로 인하여 사용하지 못했던 디바이스 하드웨어 기반 최적화 기법들을 사용하기 위해 앱 캐시 파일을 생성하는 도중에 덱스(Dex) 파일의 내용을 현재 디바이스에 맞도록 최적화하는 Dex-to-Dex 최적화가 소개되었다. 따라서, 앱 캐시 파일에 포함된 덱스 파일은 기존 앱 패키지 파일의 덱스 파일과 다른 내용을 가지게 되었다[17].

2.2 앱 캐시 파일 구조

ART는 기존의 DVM과 다르게 APK 파일이 아닌 앱 캐시 파일을 생성하고 로딩하여 앱을 실행한다. 안드로이드는 해당 디바이스에 최적화가 수행된 달빅 바이트코드를 포함하며 ELF 파일 포맷을 기반으로 확장한 OAT 파일 포맷으로 앱 캐시 파일을 구현했다. 따라서 ELF를 기반으로 동작하는 기존의 시스템의 큰 변경 없이 ART를 구현할 수 있었으나 안드로이드 버전에 따른 추가적인 기능을 확장으로 하위호환성을 포기하게 되어 버전별로 다양한 형태의 OAT 파일 포맷을 가지며 현재 안드로이드 Oreo 기준으로 138 버전을 사용하고 있다[15,16].

OAT 파일 포맷은 .oatdata 세그먼트와 .oatexec 세그먼트로 크게 2개의 세그먼트로 이루어져 있다. ELF의 .rodata 동적 심볼과 같은 파일 오프셋을 가지는 .oatdata 세그먼트는 최적화된 달빅 바이트코드를 포함한 모든 메타 정보를 포함하고 있으며 .oatexec 세그먼트는 ART의 최종 컴파일 목표인 네이티브 코드에 대한 실제 어셈블리 코드와 관련 메타 정보로 구성되어있다. 최신의 ART는 MIPS, MIPS 64, X86, X86 64, Arm, Arm 64 및 thumb2의 7가지 유형의 아키텍처를 지원한다. 즉, 같은 Android 플랫폼 버전과 같은 안드로이드 앱이라고 하더라도 설치된 디바이스의 아키텍처에 따라 생성된 앱 캐시 파일의 내용이 다름을 의미한다.

2.3 앱 캐시 파일의 로딩 및 검증 과정 취약점 분석

사용자가 디바이스의 앱을 선택하여 실행하면 ART는 앱 캐시 파일의 존재 유무를 확인한 이후에 존재한다면 메모리에 로딩하기 위한 일련의 검증 과정을 수행한다. Fig.1. App cache verification process는 안드로이드가 앱 캐시 파일을 로딩하는 중에 해당 앱 캐시 파일에 대해 검증하는 과정을 표현한 것으로, 로딩하고자 하는 앱 캐시 파일이 앱 패키지 파일에서 생성된 것인지를 확인하기 위한 과정과 현재 디바이스에서 사용되는 안드로이드 프레임워크에 맞게 생성되었는지 확인하기 위한 과정으로 총 2 단계로 진행된다.

2.1.4 Dex-to-Dex 최적화에서 밝힌 것과 같이 안드로이드는 앱 패키지 파일을 통해 앱을 설치하는 도중에 앱 캐시 파일을 생성하는데 이때 앱 캐시 파일에 포함된 덱스 파일은 앱 패키지 파일의 것과는 달리 최적화에 의해서 내부 달빅 바이트코드가 변경된다[17]. 따라서 앱 캐시 파일의 덱스 파일로부터 더 이상 앱 패키지 파일의 덱스 파일과 비교할 수 있는 체크섬 값을 계산할 수 없기 때문에 앱 패키지 파일과는 달리 앱 캐시 파일에 관련된 덱스 파일에 해당하는 체크섬 값을 저장하기 위한 OatDexFile 구조체를 두고 있다. 따라서 안드로이드의 검증 방식은 앱 캐시 파일의 달빅 바이트코드와, 기준이 되었던 APK 파일의 달빅 바이트코드가 같은지를 검사하는 것이 아니라, 현재 디바이스의 앱 캐시 파일이 dex2oat를 통해 생성되었는지 수준에서만 검증을 진행한다. ART는 base.apk의 class*.dex에 대해 CRC32 checksum을 계산한 다음, 앱 캐시 파일의 OatDexFile에 존재하는 checksum 값과 일치하는지 먼저 비교한다. 이 비교를 통해 앱 캐시 파일인 base.odex가 현재의 APK에서 파생되었음을 간접적으로 확인할 수 있다. CRC32검사가 완료되면,

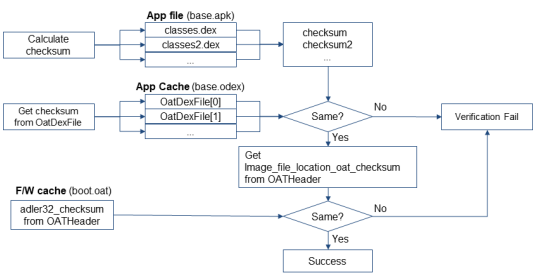

Fig. 1. App cache verification process

base.odex의 OATHeader의 image_file_location_oat_checksum 변수의 값을 꺼내 boot.oat라는 이름의 안드로이드 프레임워크 캐시 파일 내 존재하는 OATHeader의 adler32_checksum과 비교한다. 이 과정을 통해 base.odex가 해당 안드로이드 프레임워크에서 dex2oat를 통해 생성되었음을 간접적으로 알 수 있다.

결국, 공격자가 앱 캐시 파일인 OAT 구조에 잘 알고 있다면 앱 캐시 파일의 내부 달빅 바이트코드가 변조되어 있더라도 앱 캐시 파일의 정당성을 검사하는 로직을 우회하기 위해서 특정 checksum 값을 올바른 것으로 위/변조하여 우회할 수 있으므로, 앱 캐시 파일이 dex2oat를 통해 생성되었는지만을 검사하는 현재의 앱 캐시 검증 프로세스는 매우 취약하다.

III. 앱 캐시 변조 공격 실험

안드로이드 ART 환경에서 앱 캐시 변조를 통해 앱의 동작을 변조하는 공격은 2016년 Sabanal에 의해서 소개되었다. 이 공격에서 공격자는 공격 대상 앱 샌드박스를 무효로 할 수 있는 시스템 권한을 가지는 것을 전제로 한다. 이런 가정이 가능한 실제 상황은 쉽게 찾아볼 수 있다. 현재도 많은 사용자는 비인가 앱의 사용이나 안드로이드의 테마 적용 등을 위해서 스스로 루팅을 통해 시스템 권한을 획득하는 경우가 많다. 이런 경우 구글 플레이스토어와 같은 공식 앱스토어가 아닌 서드파티 앱스토어를 사용하게 되며, 악성 코드가 포함된 비인가 앱을 통해 원격 접속 등과 같은, 공격자가 디바이스에 접근할 수 있는 상황을 벌어질 수 있다. 본 연구에서의 앱 캐시 변조 공격도 이러한 상황임을 가정한다.

3.1 공격자 모델

앱 캐시 파일 변조를 수행할 수 있는 공격자의 종류는 디바이스 소유자와 원격 공격자로 크게 2가지로 구분되며 공격자의 종류 별 구분한 기준과 예상 공격 시나리오는 다음과 같다.

디바이스 소유자: 디바이스 소유자는 루팅된 디바이스를 소유하고 있으며, 앱 캐시 파일 공격을 통해 APK 파일에 대한 보호 솔루션을 가지고 있는 특정 앱에 대한 동작 변조를 할 수 있다. 이를 통해서 비용을 지불해야만 활성화 할 수 있는 기능을 임의로 활성화하거나 광고를 삭제하는 등의 이득을 꾀할 수 있다.

원격 공격자: 원격 공격자는 공격 대상 디바이스에 대한 원격 루트쉘을 가지고 있으며, 공격 대상 디바이스에서 사용자의 개인정보를 포함한 정보 취득을 위해 SNS나 메시저와 같은 앱을 대상으로 앱 캐시 파일을 변조할 수 있다. 공격을 당한 사용자는 변조 사실을 알 수 없고, 앱을 재설치하지 않았기 때문에 별도의 과정없이 기존 앱을 그대로 사용함으로써 정보가 유출될 수 있다.

3.2 공격 과정 설명

안드로이드 앱 캐시 변조 공격에 대한 공격 과정을 Fig.2. Attack process에 표현하였다. ① 공격자는 공격 대상 디바이스의 공격 대상 앱의 샌드박스에 접근해서 앱의 APK 파일과 앱 캐시 파일을 획득할 수 있으며, ② Apktool과 같은, 공개된 디컴파일 도구를 통해 앱을 디컴파일하고 ③ Smali 언어를 통한 코드를 삽입하거나 기존 코드를 변조하여 공격 대상 앱의 동작을 변조한다. 본 연구에서는 안드로이드의 토스트 메시지를 띄우는 작은 Smali 코드를 주입하는 방식으로 공격 가능성을 보이고자 한다. 사용된 Smali 코드는 Table 1. Payload for Smali code injection과 같다.

④ Apktool을 통해서 변조되어 다시 컴파일된 APK 파일은 ⑤ 디바이스의 dex2oat를 사용해서 앱 캐시 파일을 생성한다. ⑥ 공격자는 앱 캐시 파일 로딩 및 검증 과정의 취약점을 이용하기 위해서, 원본 앱 캐시 파일에 존재하는 올바른 체크섬 값을 추출하여 현재 생성한 앱 캐시 파일에 덮어씀으로써 ART는 앱 캐시 파일의 변조를 탐지하지 못하고 정상적으로 로딩 및 실행하게 된다.

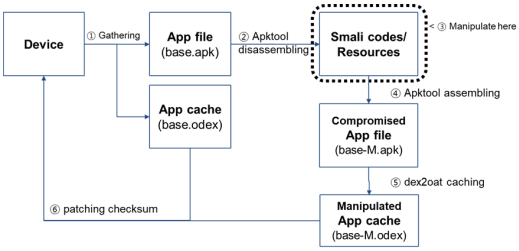

Fig. 2. Attack process

Table 1. Payload for Smali code injection

```
const-string v6, "Attacked"
const/4 v7 , 0x0
invoke-static {p0 , v6,v7},Landroid/widget/T
oast;->makeText(Landroid/content/Context;Lj
ava/lang/CharSequence;I)Landroid/widget/To
ast;
move-result-object v6
invoke-virtual {v6}, Landroid/widg
et/Toast;->show()V
```

3.3 실험 대상 앱 선정 기준

현재 서비스되고 있는 안드로이드 기반 앱 중 금융 등을 포함한 민감한 개인정보를 다루고 있으며 구글 플레이스토어에서 2018년 9월 1일 기준 민감한 개인정보를 사용할 수 있는 카테고리를 선정하여, Finance를 포함한 총 10개 카테고리에서 다운로드 수가 가장 높은 앱들을 취합하여 총 29개를 공격 대상으로 삼았다.

3.4 실험 환경

본 실험은 Apktool 2.3.4를 바탕으로 실험 디바이스 구글 넥서스 5X에서 루팅 된 안드로이드 7.0 환경에서 진행하였다. 안드로이드 앱 캐시 파일에 대한 변조 공격은 크게 다르지 않으나, 본 실험에서 진행한 방법은 앱 캐시 파일 중 OAT 파일 포맷을 가지며 odex 확장자를 가진 상황을 가정으로 설명되었다. 따라서 본 실험에서 진행한 안드로이드 7.0을 포함한 안드로이드 6.0.x 이후부터 안드로이드 7.1.x 버전까지 본 공격 과정을 통해 공격이 가능하다.

3.5 실험 결과

본 실험 결과는 Table 2. Result of App cache manipulation attack에 그 결과를 작성 하였으며, 각 항목 별로 구분하여 설명하면, 먼저 Decompiling 항목은 Apktool을 사용하여 대상 앱에 대한 디컴파일 성공 여부를 나타내며, Repackaging 항목은 Apktool을 통해 다시 APK 파일로 컴파일 성공 유무를 나타내고, Smali code injection 항목은 표 1에서 나타낸 Smali 코드를 주입하여 "Attacked"라는 토스트 메시지를 확인한 결과를 나타낸다.

실험을 진행한 결과 Smali 코드 주입 후 재생성한 앱의 캐시 파일을 원본 앱 캐시 파일과 교체함으로써 23개의 앱에 대해서 공격이 성공적으로 수행되었음을 확인했다. Decompiling 항목은 모두 O 표

Table 2. Result of App cache manipulation attack

Category	Name	Apktool		Smali code injection
		Decompiling	Repackaging	
Finance	PayPal	O	O	O
	Bank of America Mobile Banking	O	O	O
Password	Google OTP	O	O	O
	Microsoft Authenticator	O	O	O
	Blizzard Authenticator	O	O	O
	1password	O	O	O
Medical	Texas Health MyChart	O	O	O
Dating	Zoosk Dating App	O	O	O
	OkCupid Dating	O	O	O
Communication	Facebook Messenger	O	O	X
	Facebook Messenger-lite	O	O	X
	Whatsapp Messenger	O	O	O
	GMail	O	O	O
	Microsoft Outlook	O	O	O
SNS	Facebook	O	△	△
	Facebook-lite	O	△	△
	Instagram	O	X	X
	Twitter	O	X	X
	Pinterest	O	O	O
Business	LinkedIn	O	O	O
	Indeed Job Search	O	O	O
Location	Booking.com	O	O	O
	Airbnb	O	O	O
Productivity	Dropbox	O	O	O
	Google Drive	O	O	O
	Microsoft OneDrive	O	O	O
	Evernote	O	X	X
IoT	Amazon Alexa	O	O	O
	SmartThings	O	X	X

기로 Apktool 2.3.4에 의해서 디컴파일에 실패하는 경우는 없었다. Repackaging 항목에서 X 표시된 Instagram, Twitter, Evernote, SmartThings 앱은 디컴파일은 문제가 없었으나 변조하고자 한 메소드의 레지스터 개수가 16개 이상일 경우, 레지스터 개수 이상 오류를 반환하며 컴파일에 실패하였다. 현재 해당 이슈는 Apktool의 개발 사이트에서도 공유되었으나 현재까지 처리되지 않고 있다. Repackaging 항목에서 △ 표시된 Facebook과 Facebook-lite 앱의 경우 리소스 보호 솔루션의 사용으로 컴파일에 실패했던 경우이나, 본 논문에서 제안하는 리소스 보호 기법 우회 공격을 통해 디컴파일 및 변조된 앱 캐시 파일의 재 생성과 Smali 코드 주입 공격 모두 성공한 경우를 말한다. Repackaing은 가능했으나 Smali code injection에 실패한 앱은 Facebook Messenger와 Facebook Messenger-lite 2개의 앱으로 해당 앱들은 자체 앱의 코드 영역을 패킹 서비스를 사용하여 보호한 케이스로, 실험 과정의 공격 대상으로 삼은 스타트업 액티비티 코드가 보호되어 있어 디컴파일을 통해 실험에 사용한 Smali의 동작을 실험 중 확인할 수 없었기 때문에 공격에 실패한 것으로 간주했다.

변조된 캐시 파일은 표 1에서 보인 것과 같은 Smali 코드의 추가 주입으로 크기가 증가하며 이 증가량은 주입된 Smali 코드의 양과 비례한다. 결과적으로 앱의 보안 솔루션의 영향 및 Apktool의 버그 등의 이유로 일부 앱의 공격이 실패하였으나, 금융 앱을 포함한 대부분 앱에 대한 앱 캐시 변조 공격이 유효하였음을 볼 수 있었다. 캐시 파일이 변조되었을 때 사용자에게 알리지 않는 안드로이드 시스템도 문제지만, 앱 캐시 공격의 가장 큰 문제점은 기존 앱 변조 공격의 경우[9], 앱의 재설치가 필요했지만, 앱 캐시 파일 변조 공격은 앱의 재설치 없이

기존의 앱 데이터 공간에 저장된 사용자의 로그인 정보 및 앱의 메타 정보들이 그대로 노출될 수 있다는 점이 해당 공격의 가장 큰 효과이며 위협이다.

IV. 확장된 앱 캐시 변조 공격 설계

4.1 리소스 보호 기법 우회 공격

Sabanal이 제안한 간단한 공격 방법으로도 대부분의 앱에 대한 앱 캐시 파일 변조 공격을 수행할 수 있었으나, 일부 앱은 공격에 실패했다. Table 2. Result of App cache manipulation attack의 실험 결과 중 리소스 보호 솔루션에 의해서 디컴파일 자체가 불가능했던 앱에 대해서 추가적인 분석을 진행한 결과, 일부 앱에서 AndResGuard[18]와 같은 Apktool이 대상 앱을 디컴파일하는 것을 방어하기 위해 특정 문자열을 포함한 리소스 파일 이름을 가지도록 앱을 조작하는 솔루션을 사용한 것으로 확인되었다.

이전 공격에서도 확인한 것처럼 안드로이드 메니페스트 파일을 통해서 앱의 초기 구동에 사용되는 액티비티를 찾아내는 것과 같이 안드로이드 메니페스트 파일이나 string.xml 등의 리소스 정보들도 공격자가 앱의 소스코드 분석 시, 활용할 수 있기 때문에 보호가 필요하다.

이와 같은 리소스 보호 기법을 우회하기 위해서는 Apktool을 이용해서 리소스와 소스코드를 함께 디컴파일하지 않고 리소스와 소스코드를 따로 디컴파일한다. 공격을 위해 실제 수정해야 하는 것은 코드 영역이기 때문에 리소스를 제외한 코드 영역만 디컴파일 할 수 있으나 앱의 분석을 위한 안드로이드 매니페스트 파일의 획득 역시 필요하기 때문에 우선 리소스만 별도로 디컴파일하여 필요한 정보를 획득 한 후, 리소스 보호 기법으로 인해서 다시 컴파일될 수 없는 정보들을 사용해서 악성 코드의 주입 위치나 기존 코드의 변조를 위한 앱의 동작을 분석한다. 이후 해당 정보를 바탕으로 소스코드만 디컴파일된 Smali 코드들을 바탕으로 변조를 진행할 경우 리소스 보호 기법과 관계없이 앱 캐시 파일 변조를 수행할 수 있다. 본 우회 기법을 통해서 Facebook과 Facebook-lite 2개의 앱에 대해서 리소스 보호 기법을 우회하여 앱 캐시 파일 변조 공격을 진행할 수 있었다.

4.2 앱 캐시 파일 내 달빅 바이트코드 변조 공격

리소스 보호 기법과 달리 소스코드의 디컴파일 방지를 위한 방지 기법을 사용한 앱은 Apktool을 통해 쉽게 소스코드 변조를 수행할 수 없다. 따라서, 디컴파일러를 사용해서 공격 대상 앱을 분석 및 변조하는 것을 공격의 기본으로 사용하는 기존 공격 방법으로는 앱 캐시 파일 변조 공격을 완수 할 수 없었다. 이런 경우 디컴파일 과정을 통해 앱 캐시 파일을 재생성하는 대신 공격 대상 디바이스에서 추출한 앱 캐시 파일을 직접 수정할 수 있다.

다만 이전 공격에서는 APK 파일을 기반으로 디바이스의 dex2oat를 사용해서 변조된 앱 캐시 파일을 새로 생성했기 때문에 앱 캐시 파일의 컴파일 수준에 대해서 고려하지 않았으나, 앱 캐시 파일 직접 수정할 경우, 네이티브 코드 포함 여부에 따라 변조 방식을 달리해야 한다. 앱 캐시 파일에는 덱스 파일과 함께 네이티브 코드가 같이 포함된다. 2장에서 설명했던 바와 같이, ART는 앱의 실행 중 자주 호출되는 메소드에 대해서만 프로파일에 기재하고 AOT 컴파일러인 dex2oat를 사용해서 달빅 바이트코드를 디바이스의 아키텍처에 맞는 네이티브 코드로 컴파일 하게 된다. 따라서 앱 캐시 파일을 직접 수정할 때는 변조하기 위한 대상 메소드의 현재 컴파일 상태를 확인해야 한다. 대상 메소드가 네이티브 코드로 최종 컴파일 되어있을 경우 다음 장에서 설명하는 것과 같이 네이티브 코드를 직접 수정해야 하며, 아직 네이티브 코드로 컴파일 되어 있지 않을 경우 달빅 바이트코드를 수정해야 한다. 변조 공격 대상 메소드가 네이티브 코드로 컴파일 되어있는 지에 대한 여부는 변조 공격 대상 메소드를 가지는 클래스 정보를 추출 후 해당 정보를 가지고 있는 bitmap 필드 값을 통해 알 수 있다.

앱 캐시 파일이 사용하고 있는 OAT 파일 구조의 OatClass 구조체는 모든 클래스와 메소드에 대해서 네이티브 코드로 최종 컴파일 되어 있는지를 나타내는 비트맵 정보가 있으며 이 값을 통해 공격 대상 메소드의 달빅 바이트코드와 네이티브 코드 중 어느 부분을 수정해야하는지 알 수 있다. 공격 대상 메소드의 수정을 위해 달빅 바이트코드를 수정해야 한다면, 앱 캐시 파일 내에서 해당 메소드의 파일 오프셋이 어떻게 되는지 계산해야만 찾을 수 있다. 해당 메소드의 달빅 바이트코드의 실제 위치는 .oatdata 동적 심볼의 파일 오프셋과 목표 메소드가 포함된 클래스에 해당하는 OatClass 자료 구조에서 획득 가능한 대상 클래스

및 메소드의 오프셋을 더해서 찾을 수 있다.

4.3 앱 캐시 파일 내 네이티브 코드 변조 공격

공격 대상 메소드가 최종 컴파일되어 앱 캐시 파일에 네이티브 코드가 존재하며 ART에 의해서 실행될 경우에 공격자는 이전 장에서 수행한 것과는 다르게 달빅 바이트코드가 아닌 네이티브 코드의 파일 오프셋을 계산해야 한다.

대상 메소드의 네이티브 코드의 파일 오프셋을 찾는 방법은 다음과 같다. QuickMethod는 OAT 파일 구조에서 모든 메소드의 네이티브 코드에 대한 메타 정보와 실제 네이티브 코드를 가지고 있기 때문에 .oatdata 동적 심볼의 파일 오프셋, 대상 메소드와 관련된 QuickMethod 구조체의 파일 오프셋, 그리고 QuickMethod 구조체 안의 codeoffset 필드 값을 더함으로써 파일 오프셋을 구할 수 있다.

이처럼 공격 대상 앱의 앱 캐시 파일을 직접 수정하는 공격의 경우 공격자가 앱 캐시 파일의 내부 구조를 자세히 알아야 하며, 변조된 앱 캐시 파일을 실행하기 위해서는 이전 공격과는 다르게 체크섬 값 위조뿐만 아니라 추가된 달빅 바이트코드나 네이티브 코드에 따라 앱 캐시 파일 내 모든 관련된 오프셋 값을 실제 파일 오프셋과 같도록 변조해야 한다.

V. 방어 기법 제안

이전 장에서 보인 것과 같이 앱 캐시 파일의 변조 공격은 여전히 유효하며 방어 기법의 연구가 시급하다. Jiawan[19]은 안드로이드 앱 캐시 변조 공격에 대응하기 위해 OAT 컴파일러를 활용하여 앱 캐시 파일에 대한 무결성 검사 방법은 제안하였다. 이 방법을 통해 앱 캐시 파일에 포함된 덱스 파일에 대한 최적화인 Dex-to-Dex 최적화를 포함한 최적화 기법들에 대응할 수 있었지만, 이 방법을 적용하기 위해서는 검사 기준이 되는 데이터를 생성하기 위해 필요한 앱 캐시 파일의 분석이 필요하나, 안드로이드 버전에 따라 크게 달라지는 OAT 파일 포맷의 가변성과 복잡한 구조를 가진 특성때문에 적용이 쉽지 않다. 무엇보다 ART에서 수행하는 최적화의 최종점은 네이티브 코드로의 컴파일을 통한 성능 최적화임에도 불구하고 달빅 바이트코드에 대한 무결성 검사만 가능한 것이 가장 큰 제약 사항이다.

Jiawan[19]의 무결성 검사 방법을 통해 보호하지 못했던 네이티브 코드에 대한 변조 공격을 막기 위한 한 가지 대안은 특정 보호 대상 메소드에 대한 네이티브 코드에 대해서 시스템의 AOT 컴파일러와 컴파일러 필터를 이용하여 얻어내는 것이다. 모든 메소드에 대해서 네이티브 코드로의 컴파일을 유도하는 컴파일러 필터를 사용하여 모든 메소드들에 대한 네이티브 코드를 획득하고, 런타임에 수행되는 메소드가 네이티브 코드로 최종 컴파일된 메소드인지를 앱 캐시 파일의 구조 분석을 통해 확인한 뒤, 네이티브 코드로 컴파일이 되어있을 경우, 이전 단계에서 추출한 네이티브 코드와의 비교를 통해 무결성 검사를 수행하는 방법이다. 다만, 이 방법은 초기화 시, 많은 시간을 소비할 수 있으며 이는 앱 패키지 파일의 크기와 디바이스의 성능에 따라 전체 컴파일을 완료하는 시간이 달라진다.

이와 반대로 3.5. 실험 결과에서 보인 것과 같이 Apktool을 통한 디컴파일 과정만 실패하더라도 공격자가 앱 캐시 파일을 변조하는 작업은 실제 앱 캐시 파일에 대한 달빅 바이트코드의 삽입 및 파일 내 모든 오프셋 값에 대한 정비 등 공격이 복잡해진다는 것을 알 수 있었다. 따라서, 디컴파일 방지 기법을 통해 무결성 검사와 같은 여러 환경적인 제약 사항을 고려하지 않고 앱 캐시 파일 변조 공격에 대해 효과적으로 방어할 수 있을 것으로 기대한다.

VI. 결 론

본 논문에서는 2016년 공개된 ART의 앱 캐시 파일 변조 공격의 유효성 및 공격의 파급력을 확인하기 위해, 공격을 직접 구현한 실험을 수행하여 총 29개 앱 중 21개 (72.4%)의 앱이 취약한 것을 보였으며 공격 시, 사용자가 그 사실을 알 수 없고, 기존의 앱 변조 공격과는 다르게 앱의 재설치가 필요 없어서, 이미 사용자에 의해 저장된 개인정보 등이 그대로 노출될 수 있음을 확인하였다. 또한 실험 결과를 분석하여, 리소스 보호 솔루션 우회, 달빅 바이트코드, 네이티브 코드 변조의 추가 공격 방안을 제시하였고, 기존 공격으로 실패한 4개 앱에 대해 공격에 성공하여, 총 29개 앱 중 25개 (86.2 %)의 앱이 안드로이드 앱 캐시 변조 공격에 취약함을 보였다.

마지막으로 이러한 공격들을 방어하기 위한 네이티브 코드 동적 생성을 통한 무결성 검사 기법 적용 및 디컴파일 방지 기법 사용을 제안하였다. 해당 방어 기법들은 서로 다른 장/단점을 가지고 있기 때문에 여러

보호 기법들을 적절하게 적용하여야 한다. 본 논문에서 제안한 디컴파일 방지 기법 역시 무결성 검사와 같은 보안성이 뛰어나지만 안드로이드의 업데이트 등 시스템 변경에 대한 대응 비용이 많이 드는 검사와 함께 함께 적용해야 하며 APK 파일 수준과 앱 캐시 파일 수준 모두에서 앱의 변조 공격을 막아야 한다.

추후에는 앱 캐시 파일과 앱 실행 메모리에 대한 무결성 검사 방법을 연구하여 파일에 대한 변조와 실행 중 메모리 변조를 통한 앱의 동작을 변조하는 공격에 대응하고자 한다.

References

[1] Symantec, "Internet security threat report internet report volume 23" https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-executive-summary-en.pdf, Accessed Feb. 11. 2019.

[2] Xuxian Jiang and Yajin Zhou, "Dissecting android malware: characterization and evolution," 2012 IEEE Symposium on Security and Privacy (SP), pp. 95-109. May. 2012.

[3] Github, "Jesusfereke/smali" https://github.com/jesusfreke/Smali, Accessed Feb. 11. 2019.

[4] Github, "Apktool" https://github.com/iBotPeaches/Apktool, Accessed Feb. 11. 2019.

[5] Rowland Yu, "Android packers: facing the challenges, building solutions," Proceedings of the 24th Virus Bulletin International Conference (VB 2014), pp.266-275, Sep. 2014.

[6] Lukas Dresel, Mykolai Protsenko, and Tilo Müller, "ARTIST: the android runtime instrumentation toolkit," 2016 11th International Conference on Availability, Reliability and Security (ARES), pp. 107-116, Sep. 2016.

[7] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky and Sebastian Weisgerber, "ARTist: The android runtime instrumentation and security toolkit," 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 481-495, Apr. 2017.

[8] Oliver Schranz, "ARTist - A novel instrumentation framework for reversing and analyzing android apps and the middleware" https://www.blackhat.com/us-18/briefings/schedule/index.html#artist---a-novel-instrumentation-framework-for-reversing-and-analyzing-android-apps-and-the-middleware-10710. Accessed Feb. 11. 2019.

[9] Sabanal, Paul, "Hiding behind ART" https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf, Accessed Feb. 11. 2019.

[10] Google, "Configuring ART" https://source.android.com/devices/tech/dalvik/configure, Accessed Feb. 11. 2019.

[11] Google, "Android 5.0 updates" https://developer.android.com/about/versions/android-5.0-changes?hl=ko, Accessed Feb. 11. 2019.

[12] Google, "Android 7.0 for developers" https://developer.android.com/about/versions/nougat/android-7.0?hl=ko#jit_aot, Accessed Feb. 11. 2019.

[13] Google, "Implementing ART Just-In-Time (JIT) compiler" https://source.android.com/devices/tech/dalvik/jit-compiler, Accessed Feb. 11. 2019.

[14] Zhong, X, "ART JIT in Android N" https://connect.linaro.org/resources/las16/las16-201/, Accessed Feb. 11. 2019.

[15] Github, "Lief-project" https://github.com/lief-project/lief, Accessed Feb. 11. 2019.

[16] Romain Thomas, "Android OAT formats" http://www.romainthomas.fr

/post/android-oat/, Accessed Feb. 11. 2019.
〔17〕 Github, "DEX-to-DEX optimization" https://github.com/anestisb/oatdump_plus, Accessed Feb. 11. 2019.
〔18〕 Github, "AndResGuard" https://github.com/shwenzhang/AndResGuard, Accessed Feb. 11. 2019.
〔19〕 Jia Wan, Mohammad Zulkernine, Phil Eisen and Clifford Liem, "Defending application cache integrity of android runtime," International Conference on Information Security Practice and Experience. Springer, Cham, pp. 727-746, Dec. 2017.

〈저자소개〉

홍 석 (Seok Hong) 정회원
2011년 2월: 조선대학교 전자전기컴퓨터공학과 졸업
2010년 12월~현재: 삼성전자 Samsung Research 재직
2018년 3월~현재: 성균관대학교 DMC공학과 석사과정
〈관심분야〉 정보보호, 시스템 보안

김 동 욱 (Dong-uk Kim) 정회원
2007년 8월: 고려대학교 산업시스템정보공학과 졸업
2007년 7월~2008년 8월: 한국생산성본부 연구원 재직
2014년 2월: KAIST 산업및시스템공학과 박사 졸업
2014년 3월~현재: 삼성전자 Samsung Research 재직
〈관심분야〉 정보보호, 시스템 보안

김 형 식 (Hyoungshick Kim) 종신회원
1999년 2월: 성균관대학교 정보공학부 학사
2001년 2월: KAIST 컴퓨터 과학과 석사
2012년 2월: University of Cambridge 컴퓨터공학과 박사
2013년 3월~현재: 성균관대학교 전자전기컴퓨터공학과 조교수
〈관심분야〉 보안공학, 사용자 인증, 모바일 보안