Journal of The Korea Institute of Information Security & Cryptology(JKIISC) ISSN 1598-3986(Print)
VOL.23, NO.4, August 2013 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2013.23.4.743

안드로이드 스마트폰 뱅킹 앱 무결성 검증 기능의 취약점 연구*

김 순 일,† 김 성 훈, 이 동 훈‡
고려대학교 정보보호대학원

A study on the vulnerability of integrity verification functions of android-based smartphone banking applications*

Soonil Kim,† Sunghoon Kim, Dong Hoon Lee‡
Graduate School of Information Security, Korea University

요 약

최근, 정상 앱에 악성코드를 추가하여 안드로이드 마켓에 재배포 되는 악성 앱들이 발견되고 있다. 금융거래를 처리하는 뱅킹 앱들이 이와 같은 공격에 노출되면 인증정보 및 거래정보 유출, 부정거래 시도 등 많은 문제점들이 발생할 수 있다. 이에 대한 대응방안으로 금융당국이 관련 법규를 제정함에 따라 국내 은행들은 뱅킹 앱에서 무결성 검증 기능을 제공하고 있지만, 해당 기능의 안전성에 대한 연구는 이루어진 바가 없어서 신뢰하기 어렵다. 본 논문에서는 안드로이드 역공학 분석 기법들을 이용하여 뱅킹 앱의 무결성 검증 기능 취약점을 제시한다. 또한, 제시한 취약점이 이용될 경우, 실제 뱅킹 앱의 무결성 검증 기능이 매우 간단하게 우회되어 리패키징을 통한 악성코드 삽입 공격이 이루어질 수 있으며 그 위험성이 높다는 것을 실험결과로 증명한다. 추가적으로, 취약점을 해결하기 위한 방안들을 구체적으로 제시함으로써 앱 위변조 공격에 대응하여 스마트폰 금융거래 환경의 보안 수준을 높이는데 기여한다.

ABSTRACT

In recent years, the malicious apps with malicious code in normal apps are increasingly redistributed in Android market, which may incur various problems such as the leakage of authentication information and transaction information and fraudulent transactions when banking apps to process the financial transactions are exposed to such attacks. Thus the financial authorities established the laws and regulations as an countermeasures against those problems and domestic banks provide the integrity verification functions in their banking apps, yet its reliability has not been verified because the studies of the safety of the corresponding functions have seldom been conducted. Thus this study suggests the vulnerabilities of the integrity verification functions of banking apps by using Android reverse engineering analysis techniques. In case the suggested vulnerabilities are exploited, the integrity verification functions of banking apps are likely to be bypassed, which will facilitate malicious code inserting attacks through repackaging and its risk is very high as proved in a test of this study. Furthermore this study suggests the specific solutions to those vulnerabilities, which will contribute to improving the security level of smartphone financial transaction environment against the application forgery attacks.

Keywords: Android, Smartphone Banking, Application Integrity, Reverse Engineering

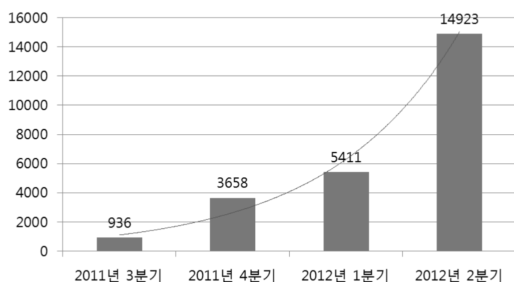
접수일(2013년 5월 21일), 수정일(2013년 7월 16일), 게재확정일(2013년 7월 17일)
* 본 연구는 방위사업청과 국방과학연구소의 지원으로 수행되었습니다.
† 주저자, sikim@kftc.or.kr
‡ 교신저자, donghlee@korea.ac.kr(Corresponding author)

I. 서 론

최근 스마트폰, 태블릿PC 등 스마트 기기의 급속한 보급과 전자금융거래의 편리성 때문에 스마트폰을 이용한 거래가 매우 빠르게 증가하고 있다. 한국은행에 따르면, 2012년 9월말 현재 국내의 스마트폰 기반 모바일뱅킹 등록고객수는 1,984만명으로 전분기말 대비 305만명(18.2%) 증가하였으며 이용건수와 금액은 1,325만건과 8,913억원으로 전분기말 대비 각각 12.1%, 12.8% 증가하였다[1]. 스마트폰 이용자가 증가함에 따라, 관련 보안위협도 계속해서 증가하고 있다. Kaspersky Lab의 보고서에 따르면, [그림 1]과 같이 안드로이드 운영체제를 대상으로 하는 신규 악성코드 수는 2012년 2분기에만 이전 분기에 비해 3배 가까이 증가한 14,900개 이상 발견될 정도로 폭발적으로 증가하고 있다[2]. 2011년에는 정상 앱을 위변조하여 악성코드를 심고 안드로이드 마켓에 재배포하여 시스템 정보 탈취 및 외부 애플리케이션 원격설치 등을 실행하는 Droid- Dream 악성 앱이 등장하기도 했다[3].

이와 같이 안드로이드 운영체제가 악성코드 작성자들의 주요 표적이 되는 이유는 다음과 같다. 안드로이드 앱은 이식성이 높은 자바 언어로 구현되어, 역공학이 쉽기 때문에 앱 위변조를 이용한 악성코드 삽입과 리패키징(repackaging)이 용이하다. 또한, 구글의 공식 마켓인 Play스토어 이외에 써드파티(third-party) 마켓이 활성화되어 있고 직접 안드로이드 애플리케이션 패키지 파일(APK, 이하 APK)을 설치하는 것도 가능하기 때문에 악성 앱의 배포가 쉽다.

안드로이드 운영체제의 문제점은 금융 앱에도 심각한 위협이 된다. 정식으로 배포된 금융 앱 안에 인증정보 및 금융정보를 유출할 수 있는 악성코드를 삽입하여 리패키징해 배포한다면 심각한 2차 피해로 이어질 가능성이 높기 때문이다. 금융당국은, 금융기관 또는 전자금융업자가 전자금융거래에서 이용자에게 제공하거나 거래를 처리하기 위한 전자금융거래프로그램의 위변조 여부 등 무결성을 검증할 수 있는 방법을 제공하도록 의무화[4]하고 있으며 이에 따라 각 금융기관들은 금융 앱에서 무결성 검증 기능을 제공하고 있다. 그러나 현재까지 스마트폰에서 서비스되는 금융 앱의 무결성 검증 기능에 대한 안전성이 연구된 사례가 없기 때문에 해당 기능의 안전성을 신뢰하기 어렵다. 또한, 스마트폰 뱅킹 앱 위변조에 대한 대응방안도 추상적인 수준의 권고에 그치고 있다[5][6].



[그림 1] 안드로이드 대상 신규 악성코드의 증가 추이

본 논문에서는 안드로이드 스마트폰 뱅킹 앱을 대상으로 안드로이드 역공학 기법[7][8][9][10]들을 이용하여 무결성 검증 기능의 취약점, 메모리에 공인인증서 비밀번호가 평문으로 남는 취약점을 분석하였다. 분석 결과, 뱅킹 앱의 무결성 검증 기능은 APK 파일 전체 내용과 서버에서 받은 난수 값을 조합하여 해시값을 생성하는 방식으로 되어있어, 검증하고자 하는 APK 파일의 경로를 원본 APK 파일로 자바 코드에서 지정하면 쉽게 우회가 가능함을 확인하였다. 이러한 취약점으로 인해 공인인증서 정보 탈취, 계좌이체 수취계좌번호의 변경과 같은 악성코드 삽입이 가능함을 확인하였다. 즉, 기존 기법은 공격자에 의한 앱 위변조 공격을 차단할 수 없으므로, 다음과 같은 대응방안을 제시한다. 자바 실행 코드를 보호하기 위하여 자바 코드의 동적 로딩, 자체 수정 코드(Self-modifying code) 기법 활용 및 소스코드 난독화 강화가 필요하고, 무결성 검증을 강화하기 위하여 메모리 무결성 검증 및 무결성 검증 지점 확대가 필요하며, 공격 시도를 차단하기 위하여 무결성 오류 발생 계정 및 접속IP 차단, 무결성 미검증 버전의 앱 사용 중지가 필요하다. 본 논문의 기여도는 다음과 같다.

- 최초로, 국내 은행들의 안드로이드 스마트폰 뱅킹 앱 무결성 검증 기능에 관한 취약점을 분석하였다.
- 자바 디컴파일, smali 코드 파일에 로그 삽입 후 실행 로그 분석, 시스템 콜 추적, 메모리 덤프(dump), 네이티브(native) 코드 라이브러리 분석 등 공격자가 사용할 수 있는 안드로이드 앱 분석 기법들을 체계적으로 정리하였다.
- 국내 주요 7개 은행들의 실제 뱅킹 앱을 대상으로 무결성 검증 우회 및 악성코드 삽입 테스트를 실시하여, 무결성 검증 취약점을 이용한 앱 위변조 공격의 위험성이 매우 높음을 보였다.

- 무결성 검증 취약점 해결방안들을 구체적으로 제시함으로써 스마트폰 금융거래 환경의 보안 수준을 높이는데 기여한다.

본 논문의 구성은 다음과 같다. 2장에서는 배경지식을 알아보고, 3장에서는 취약점을 분석하기 위한 분석도구 및 기법들에 대해서 살펴본다. 4장에서는 취약점 분석 및 공격 시나리오를 기술하며, 5장에서는 4장에서 기술한 시나리오를 토대로 국내 주요 은행들의 안드로이드용 뱅킹 앱에 대한 테스트를 수행한 결과를 기술한다. 6장에서는 본 논문에서 제기한 안드로이드용 뱅킹 앱의 취약점에 대한 대응방안을 제시하고, 7장에서는 요약과 함께 결론을 맺는다.

II. 배경지식

본 장에서는 뱅킹 앱의 취약점 분석 및 무결성 검증 기능을 우회하기 위해서 필요한 배경지식을 알아본다.

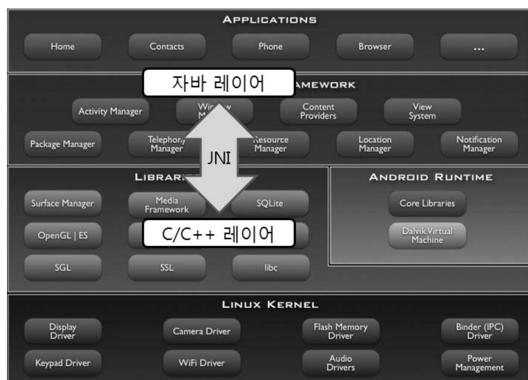
2.1 안드로이드 운영체제 구조

안드로이드는 운영체제, 미들웨어(middleware)와 주요 애플리케이션들을 포함하고 있는 모바일 장치를 위한 소프트웨어 묶음이다[11]. 안드로이드 운영체제는 [그림 2]와 같이 Applications, Application Framework, Libraries, Android Runtime, Linux Kernel 5개의 주요 컴포넌트로 구성된다. Applications 계층에는 이메일, 전화, SMS, 브라우저 등의 기본 애플리케이션들이 있으며 사용자가 개발하거나 설치한 애플리케이션들은 여기에 위치하게 된다. Application Framework 계층에서는 안드로이드 애플리케이션 개발 시에 이용할 수 있는 다양한 라이브러리들을 제공한다. Applications와 Application Framework 계층은 모두 자바 언어로 구성되어 있다. Libraries 계층에는 안드로이드 시스템에서 사용하는 C/C++ 라이브러리들이 포함되어 있다. Android Runtime 계층에는 자바 프로그래밍 언어에 필요한 코어 라이브러리와 달빅 가상머신(Dalvik Virtual Machine)이 위치한다. 안드로이드 애플리케이션은 자바 컴파일러에 의해 컴파일되고 DEX(Dalvik Executable Format) 형식으로 변환되어 자신만의 달빅 가상머신 인스턴스(instance)를 가진 별도의 프로세스에서 실행된다.

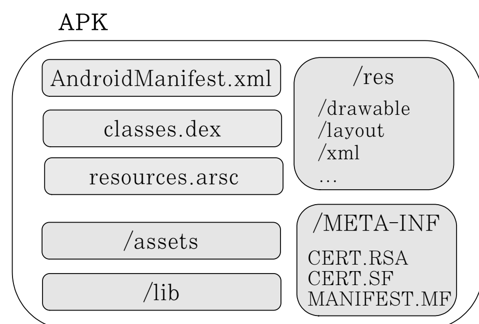
안드로이드 애플리케이션 개발자는 안드로이드 SDK(Software Development Kit)를 이용해 자바 기반의 프로그램을 작성하고 NDK(Native Development Kit)를 이용해서 자바 애플리케이션과 연결 가능한 C/C++ 기반의 네이티브 라이브러리를 작성할 수 있다. 그리고 자바에서 C/C++ 언어로 작성된 라이브러리의 특정 함수를 사용하거나 반대로 C/C++ 프로그램에서 자바 코드를 호출할 때에는 JNI(Java Native Interface)를 사용해야 한다[12]. 이와 같은 구조를 이용하여 안드로이드 앱은 무결성 검증과 같은 중요 부분은 디컴파일이 쉬운 자바 코드로 구현하지 않고 네이티브 코드로 구현하는 것이 일반적이다.

2.2 APK 파일 구조

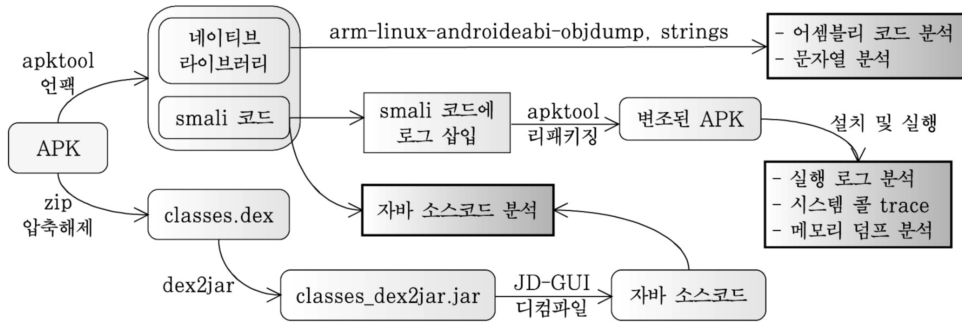
안드로이드 애플리케이션은 설치를 위한 APK 파일로 패키징되어 있으며, APK 파일은 프로그램 코드와 리소스들(resources)을 포함하는 자바 표준 jar 파일과 유사하다[13]. 일반적으로 APK 파일의 구조는 [그림 3]과 같다. AndroidManifest.xml 파일



[그림 2] 안드로이드 운영체제의 구조



[그림 3] APK 파일의 구조



〔그림 4〕 안드로이드 앱 분석 절차

은 애플리케이션 정보, 접근 권한 정보 등의 메타 정보를 포함하고 있는 XML 파일이고, classes.dex 파일은 달빅 가상머신에 의해서 로드되어 실행될 자바 바이트코드(bytecode) 정보이며, res 디렉토리는 레이아웃, 문자열 등을 정의한 파일들로 구성되어 있다〔14〕. 그리고 lib 디렉토리에는 컴파일된 네이티브 라이브러리들이 포함되고, assets 디렉토리에는 애플리케이션에서 추가로 필요한 파일들을 포함시킬 수 있으며, 해당 파일들은 android.content.res.AssetManager 클래스로 처리해야 한다.

III. 분석 도구 및 기법

본 장에서는 안드로이드 운영체제에 설치된 애플리케이션의 취약점을 분석하기 위한 도구 및 기법들을 설명한다. 전체적인 분석 절차는 〔그림 4〕와 같다. APK 파일을 apktool로 언팩한 후에 생성되는 네이티브 라이브러리는 안드로이드 NDK의 arm-linux-androideabi-objdump로 디스어셈블링(disassembling)하여 어셈블리 코드를 분석하거나 리눅스의 strings 명령어로 문자열 분석을 하고, smali 코드에는 디버그 로그 삽입 후에 apktool로 리패키징하여 변조된 APK를 만들어서 안드로이드 기기에 설치 및 실행하여 실행 로그 분석, 시스템 콜 추적, 메모리 덤프 분석을 한다. APK 파일 압축해제 후에 생성되는 classes.dex 파일은 dex2jar를 이용하여 JAR(Java ARchive) 형식으로 변환하고 JD-GUI 디컴파일 도구를 이용하여 자바 소스코드로 변환해서 소스코드 분석을 한다. 자세한 분석도구 및 기법에 대해서는 다음 절에서부터 설명한다.

3.1 APK 파일 언팩(unpack) 및 리패키징

안드로이드 앱은 APK 파일을 언팩하여 그 내용을 확인하거나 언팩된 파일들을 수정한 후에 리패키징 하는 방법을 이용하여 효과적으로 분석할 수 있다. APK 파일의 언팩 및 리패키징은 안드로이드 APK 파일 역공학 도구인 apktool을 사용하면 가능하다. apktool로 APK 파일을 언팩하면 〔그림 3〕과 같은 파일 및 디렉토리들이 추출되며, 안드로이드 DEX 형식을 디스어셈블(disassemble)한 파일들이 들어있는 smali 디렉토리가 추가적으로 생성된다. 추출된 디렉토리 및 파일들은 apktool을 이용하여 리패키징하고 자바 SDK에 포함된 jarsigner 도구를 이용하여 서명하면 안드로이드 운영체제에 설치 가능한 APK 파일로 다시 만들 수 있다.

3.2 자바 클래스 디컴파일 및 소스코드 분석

APK 파일은 JAR(Java ARchive) 파일 형식에 기반한 ZIP 압축파일 형식이므로, APK 파일을 압축해제하면 classes.dex 파일을 얻을 수 있다. classes.dex 파일은 dex2jar 프로그램을 이용하여

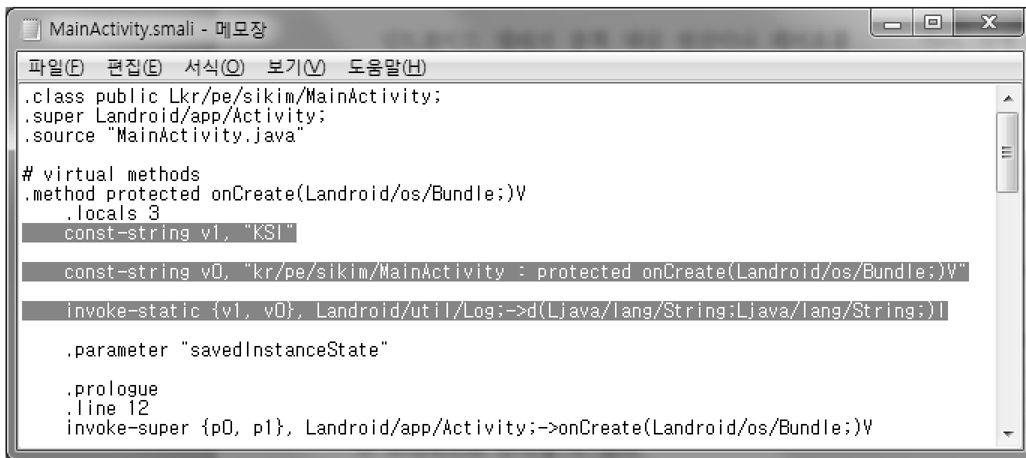
MainActivity.class

```
package kr.pe.sikim;

import android.app.Activity;

public class MainActivity extends Activity
{
  protected void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130903040);
    Toast.makeText(this, getPackageCodePath(), 1).show();
```

〔그림 5〕 JD-GUI로 디컴파일된 자바 클래스



〔그림 6〕 디버그 로그를 삽입한 smali 코드

JAR 파일로 변환이 가능하다. JAR 파일에는 자바 클래스 파일들이 포함되어 있으므로 JD-GUI와 같은 자바 디컴파일러 도구를 이용하여 〔그림 5〕와 같이 자바 소스파일로 디컴파일이 가능하다. 따라서 소스코드 수준에서 정적 분석을 하거나 소스코드 수정 후에 다시 컴파일하는 것이 가능하다.

3.3 smali 코드 및 실행 로그 분석

3.3.1 smali 코드

smali 코드는 apktool에 의한 APK 파일 언팩 과정에서 디스어셈블러(disassembler)에 의해서 생성된다. smali 파일들은 자바 소스파일과 동일한 패키지 구조를 가지고 클래스 또는 인터페이스 단위로 생성되며, 〔그림 6〕에서 보는 바와 같이 자바 소스코드 보다는 가독성이 떨어지지만 기능 및 제어흐름을 충분히 이해할 수 있는 코드로 작성되어 있다. smali 코드는 텍스트 편집기에서 쉽게 수정할 수 있고, apktool에 의한 리패키징 과정에서 어셈블러에 의해 DEX 형식의 바이트코드로 자동 변환된다. 따라서 smali 코드에서 위변조하고자 하는 코드를 쉽게 찾을 수만 있다면 smali 코드 수정을 통한 앱 위변조는 매우 쉽다.

3.3.2 smali 코드에 로그 삽입

안드로이드 앱에서 공격 대상 연산이나 제어흐름들을 찾아내는 방법은 앱 실행 시에 디버거를 이용한 동적 분석기법이 있다. 그러나 최근에는 안티 디버깅 기술들이 연구되거나 적용되고 있어서 디버깅이 쉽지 않고 제약사항이 많다. 따라서 본 논문에서는 smali 파일에 디버그 로그를 삽입하여 제어흐름을 분석하는 기법을 사용하였다. 이 기법은 smali 코드 파일들을 읽어서 모든 메소드에 자동으로 디버그 로그를 삽입하는 프로그램을 만들고, 로그 삽입을 원하는 smali 파일들에 〔그림 6〕과 같이 로그를 삽입하는 방법이다. 주요 변수들에 대한 디버그 로그를 추가하면 해당 변수들의 상태 값도 추적이 가능하다.

3.3.3 실행 로그 분석

3.3.2에서와 같이 smali 파일들에 로그를 삽입한 후에 apktool로 리패키징한 앱을 안드로이드 스마트폰에 설치 및 실행하고 안드로이드 SDK의 monitor 도구로 로그 모니터링을 하면 해당 앱의 제어 흐름을 〔그림 7〕과 같이 쉽게 파악할 수 있다. 이와 같은 로그를 이용하면 해당 앱의 특정 기능이 실행중일 때, 이를 처리하는 자바 클래스의 메소드를 쉽게 찾을 수 있으므로 소스코드 난독화가 되어 있더라도 소스코드 분석 시간을 크게 줄일 수 있다.

L.	Text
D	kr/pe/sikim/MainActivity : public constructor
D	kr/pe/sikim/MainActivity : protected onCreate(
D	kr/pe/sikim/MainActivity : public onCreateOpti

〔그림 7〕 디버그 로그를 삽입한 앱의 실행 로그

3.4 시스템 콜(system call) 분석

strace는 프로그램에 의해서 사용되는 시스템 콜이나 프로그램이 받는 시그널들을 모니터링하는 디버깅 도구이며 안드로이드 에뮬레이터에 설치되어 있다. strace를 이용하면 특정 프로세스에서 사용하는 시스템 콜을 분석할 수 있기 때문에, 모니터링 하고자 하는 안드로이드 앱에 대한 동적 분석이 가능하다. [그림 8]은 안드로이드 운영체제에서 실제 실행중인 프로세스에 대한 strace 결과의 일부분이다.

```
C:\Windows\system32\cmd.exe - adb shell
ioctl(9, 0xc0186201, 0xbe8864c8)        = 0
msgget(0x1, 0xbe8865d0, 0xbe8865d0, 0x400be4b0) = 0
getpid()                                = 552
getuid32()                              = 10040
semget(0x21, 0xbe8874e8, 0x10, 0)       = 0
msgget(0x1, 0xbe887648, 0xbe887648, 0x400be4b0) = 0
msgget(0x1, 0xbe887648, 0xbe887648, 0x400be4b0) = 0
writev(5, [{"Fu\0\0\0", 4}, {"\2\30\0\0\0\0kr.pe.sikim.MainActivity\n", 30}], 2) = 34
recv(1082601804, ptrace: umoven: I/O error
0x81, 1, MSG_PEEK|MSG_EOR|MSG_WAITALL|MSG_ERRQUEUE|MSG_CONFIRM|MSG_FIN|MSG_SYN|MSG_RST|MSG_NOSIGNAL|MSG_MORE|0xffff0000) = 1
```

[그림 8] strace를 이용한 시스템 콜 trace

3.5 메모리 덤프(dump) 분석

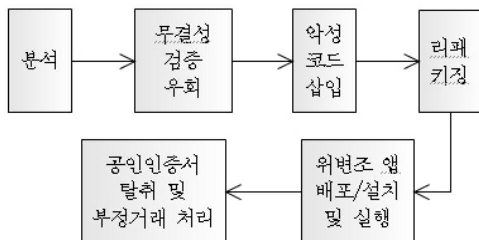
안드로이드 운영체제에서는 실행중인 앱의 힙(heap) 메모리 덤프가 가능하다. 덤프한 힙 메모리 내용은 Eclipse Memory Analyzer[15]와 같은 도구를 이용하면 쉽게 분석이 가능하다. 메모리 덤프 파일을 분석하면 해당 시점의 메모리 상태 정보를 알 수 있으므로, 공인인증서 비밀번호 등의 중요 정보가 메모리에 평문으로 저장되는지 여부 등을 알 수 있다.

3.6 네이티브(native) 코드 라이브러리 분석

네이티브 코드 라이브러리는 안드로이드 NDK에 포함된 arm-linux-androideabi-objdump 도구를 이용하여 디스어셈블링(disassembling)하고 어셈블리 코드를 분석할 수 있다. 또한, 리눅스 운영체제의 strings 명령어를 사용하여 라이브러리 파일에 포함되어 있는 문자열들을 추출하여 대략적인 기능을 추정하는 분석을 할 수 있다.

IV. 스마트폰 뱅킹 앱 위변조 공격 시나리오

본 장에서는 Ⅲ장에서 기술한 분석 기법들을 이용하여 안드로이드 스마트폰 뱅킹 앱의 취약점을 파악하고, 해당 취약점을 이용하여 앱 무결성 검증을 우회하는 방법을 기술한다. 무결성 검증이 우회되면 smali 코드 수정을 통해서 악성코드를 삽입하더라도 뱅킹 서버에서 위변조된 앱으로 탐지하지 못한다. 이를 이용하여, 악성코드가 삽입된 위변조 앱이 정상적인 뱅킹 앱처럼 동작하면서 인증 정보를 탈취하거나 수취계좌를 변조하여 부정거래를 발생시키는 해킹 시나리오를 구성해 보았다.



[그림 9] 앱 위변조 공격 시나리오

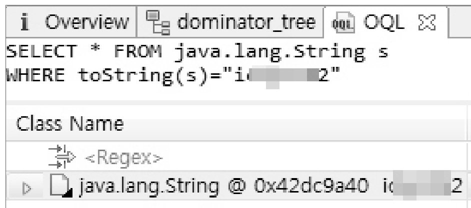
4.1 앱 무결성 취약성

4.1.1 분석

먼저, 뱅킹 앱에서 무결성 검증을 수행하는지 여부 및 어느 지점에서 무결성 검증이 처리되는지 분석한다. 안드로이드 운영체제에 설치된 APK 파일을 PC로 복사하여 언팩하면 자바 클래스 및 인터페이스 파일 단위로 smali 어셈블리 파일이 생성된다. 직접 작성한 로그 삽입 프로그램을 이용하여 모든 smali 코드 파일들의 메소드 내에 디버그 로그를 자동으로 삽입하고 apktool을 이용하여 리패키징한다. 리패키징한 앱을 안드로이드 운영체제에 설치하고 실행 시, 위변조 관련 오류가 발생하고 종료된다면 앱 무결성 검증을 수행하고 있음을 알 수 있다. 이 때, 앞에서 삽입한 디버그 로그를 추적하여, smali 코드와 디컴파일된 자바 소스코드에서 무결성 검증 기능이 수행되는 자바 클래스의 메소드를 알아낸다.

```
)haI
HBJ7
y7<s
getApplicationInfo
android/content/pm/ApplicationInfo
()Landroid/content/pm/ApplicationInfo;
/proc/
/cmdline
```

[그림 10] 네이티브 라이브러리의 strings 결과



〔그림 11〕 메모리의 공인인증서 비밀번호 평문

다음으로, smali 코드와 자바 소스코드 및 시스템 콜 추적 등을 분석하여 무결성 검증 대상이 APK 파일인지 확인하고, 검증 대상이 APK 파일이라면 해당 파일에 접근하기 위해서 현재 실행중인 앱의 APK 파일 경로를 어떻게 얻어오는지 확인한다. 일반적으로, 현재 실행 중인 앱의 APK 파일 경로는 안드로이드 자바 API에서 제공하는 android.-content.Context 클래스의 getPackageCode-Path() 또는 getApplicationInfo() 메소드를 통해서 얻을 수 있다. 따라서 네이티브 코드에서 실제 무결성 검증용 해시값을 생성하더라도 APK 파일의 경로를 구하기 위해서는 자바 코드를 이용해야 한다. 자바 코드에서 APK 파일의 경로를 구해서 네이티브 라이브러리 함수 호출시에 인자로 넘기는 방법과 네이티브 라이브러리 함수에서 자바 클래스의 메소드를 호출하여 APK 파일의 경로를 구하는 방법, 2가지를 생각해 볼 수 있다. 만약, 네이티브 라이브러리 함수에서 JNI를 이용하여 자바 클래스의 메소드를 호출하는 경우라면, 네이티브 라이브러리의 어셈블리 코드를 분석하거나 〔그림 10〕과 같이 리눅스의 strings 명령어를 이용하여 어떤 자바 클래스나 메소드를 사용하는지 확인 가능하다.

마지막으로, 공인인증서 로그인 시에 뱅킹 앱의 힙(heap) 메모리를 덤프하여 공인인증서 비밀번호가 메모리에 평문으로 존재하는지 여부를 확인한다. 〔그림 11〕과 같이 비밀번호가 메모리에 평문으로 존재한다면 평문 비밀번호의 탈취가 가능하다.

4.1.2 우회

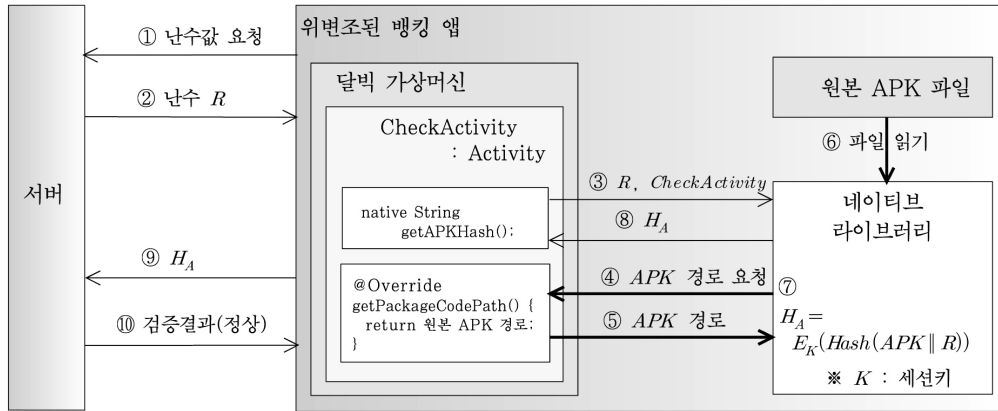
정식으로 배포된 안드로이드 앱을 위변조하고 리패키징하면 APK 파일이 배포된 원본과 다르게 변경된다. 그러므로 스마트폰에서 실행중인 앱의 APK 파일 내용에 대한 해시값을 계산하여 서버로 안전하게 전송한 후, 서버에서 보관중인 원본 APK 파일의 해시값과 비교하는 방식으로 앱 무결성 검증이 가능하다. 이러한 APK 파일을 이용한 무결성 검증의 경우, 실행을 위해서 달빅 가상머신에 로드된 APK 파일과 무결성 검증용 해시값을 계산하기 위해서 앱에서 읽은 APK 파일이 일치해야 한다. 즉, 실행코드에 대한 자체 무결성 검증이기 때문에 달빅 가상머신에 로드되어 실행중인 자신의 APK 파일 경로를 정확하게 얻어서 무결성 검증을 수행해야 한다.

안드로이드는 앱을 실행할 때마다 해당 앱의 APK 파일에 포함된 자바 코드를 달빅 가상머신에 로드하여 처리하고, 로드된 APK 파일의 경로는 android.content.Context 클래스의 getPacka-geCodePath() 또는 getApplicationInfo() 메소드 API를 통해서 실행중인 앱에서 얻을 수 있도록 제공한다. 실행중인 앱은 이러한 안드로이드 API 메소드들을 이용하여 자신의 APK 파일 경로를 얻어올 수 있는데, 자바 코드를 수정하면 해당 API 메소드들의 반환값을 변경할 수 있다. 따라서 APK 파일을 이용한 무결성 검증 시, 무결성 검증 대상 APK 파일의 경로를 구하는 자바 코드를 달빅 가상머신에 로드된 APK 파일이 아닌 원본 APK 파일의 경로를 반환하도록 수정함으로써 무결성 검증 우회가 가능하다. 실제로는 위변조된 APK 파일이 실행되지만 무결성 검증은 별도의 디렉토리에 보관된 원본 APK 파일을 읽어서 처리하기 때문에 무결성 검증용 해시값이 정상 앱과 동일하게 생성되어, 서버에서는 위변조된 앱을 정상 앱으로 판단하게 하는 방식이다. 이를 위해서 다음과 같이 정식 배포된 원본 APK 파일 자체를 위변조된 앱에 포함시키고 smali 코드 수정을 통해서 무결성 검증 대상 APK 파일의 경로를 원본 APK 파일의 경로가 되도록 변경하여 리패키징한다.

먼저, 변조되지 않은 원본 APK 파일을 변조할 앱의 assets 디렉토리에 추가하고 해당 앱이 실행될 때 원본 APK 파일을 자신의 데이터 디렉토리에 복사하

〔표 1〕 구현 방식에 따른 APK 경로 수정 방법

구현 방식	APK 경로 수정 방법
자바 코드에서 APK 경로를 구해서 네이티브 함수의 인자로 넘김	경로 값을 인자로 넘기기 전에 자바 코드에서 변경
네이티브 코드에서 자바의 getPackageCodePath() 메소드 호출	getPackageCodePath() 메소드 오버라이딩하여 반환 값 변경
네이티브 코드에서 자바의 getApplicationInfo() 메소드 호출	getApplicationInfo() 메소드 오버라이딩하여 반환 값 변경



〔그림 12〕 getPackageCodePath() 메소드 오버라이딩을 이용한 무결성 검증 우회

도록 구현한다. 다음으로, 4.1.1의 분석 결과에 따라서 〔표 1〕과 같이 smali 코드를 수정한다. 자바 코드에서 APK 파일의 경로를 구해서 네이티브 라이브러리 함수 호출시에 인자로 넘기는 방법을 사용했다면 자바 코드에서 해당 인자 값을 매우 쉽게 변경할 수 있다. 그렇지 않고, 네이티브 라이브러리 함수에서 자신을 호출한 자바 클래스의 메소드를 호출하여 APK 파일의 경로를 구하는 방법을 사용했다면 해당 메소드를 오버라이딩하여 원본 APK 파일의 경로를 반환하도록 구현하면 된다. 메소드 오버라이딩을 이용한 무결성 검증 우회는 〔그림 12〕와 같이 처리가 가능하다.

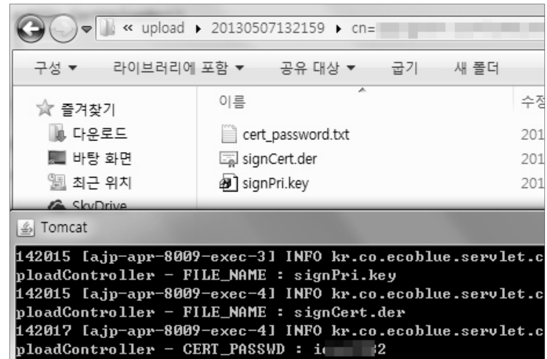
4.2 앱 위변조 공격

4.2.1 악성코드 삽입 및 리패키징

무결성 검증을 우회한 뱅킹 앱에 로그인 시에 공인인증서, 암호화된 개인키 파일 및 공인인증서 비밀번호를 해커의 웹서버로 전송하는 악성코드와 계좌이체 시에 수취계좌를 해커의 계좌번호로 변경하는 악성코드를 삽입한다. smali 코드를 수정하여 악성코드를 쉽게 추가할 수 있으며, 리패키징한 후에 안드로이드 운영체제의 스마트폰에 설치할 수 있도록 배포한다.

4.2.2 인증정보 탈취 및 부정거래 시도

악성코드를 추가한 뱅킹 앱이 안드로이드 운영체제의 스마트폰에서 정상적으로 실행되면서, 로그인 시에 해커의 웹서버로 공인인증서, 개인키 파일 및 공인인증서 비밀번호가 〔그림 13〕과 같이 정상적으로 전송



〔그림 13〕 해커의 웹서버로 전송된 공인인증서 정보

되는지 확인한다. 그리고 계좌이체 시에 해커가 변조한 계좌로 정상적으로 입금이 되는지 확인한다.

V. 실험 및 결과

국내 주요 7개 은행의 안드로이드 스마트폰 뱅킹 앱을 대상으로 Ⅳ장의 공격 시나리오를 토대로 실험한 결과는 〔표 2〕와 같다. 7개 은행은 국내의 시중은행 6개와 특수은행 1개를 무작위로 선정하였다.

실험 결과, 6개 은행은 APK 파일에 대한 언팩 및 리패키징이 가능했으며 그 중에 3개 은행은 로그인시에 앱에 대한 무결성 검증을 수행하고 있었다. 그러나 무결성 검증을 수행하는 3개 은행의 뱅킹 앱들은 무결성 검증 기능이 쉽게 우회되었다. 무결성 검증을 하지 않거나 무결성 검증이 우회된 A~F 은행들은 공인인증서 파일 및 비밀번호의 탈취가 가능했으며, 탈취한 공인인증서를 이용한 계정도용이 가능했다. 실험에서는 공인인증서 정보만 탈취했으나, 앱 위변조 공격은

〔표 2〕 국내 7개 은행 안드로이드 스마트폰 뱅킹 앱 테스트 결과

은행	리패키징 가능 여부	자바 디컴파일 가능 여부	무결성 검증		공인인증서 탈취			계좌이체 시, 수취계좌 변경
			검증 여부	우회 여부	인증서 탈취	비밀번호 탈취	탈취한 인증서로 인증	
A은행	○	○	○	○	○	○	○	○
B은행	○	○	×	-	○	○	○	○
C은행	○	○	○	○	○	○	○	○
D은행	○	○	○	○	○	○	○	○
E은행	○	○	×	-	○	○	○	○
F은행	○	○	×	-	○	○	○	○
G은행	×	○	-	-	-	-	-	-

소스코드를 수정하는 방법이기 때문에 실제로는 거의 모든 입력정보를 탈취할 수 있을 것으로 판단된다. 또한, 실험에서 사용한 위변조 앱은 정상적인 앱과 동일한 기능을 하면서 악성코드를 실행시킨다는 점에서 사용자가 인지하기 어렵기 때문에 그 위험성이 매우 크다는 것을 알 수 있었다.

G은행은 DEX 파일 형식에서 자바 클래스명을 255자 이상으로 조작하는 방법으로 역공학을 방지하고 있어서 apktool로 언팩 시에 오류가 발생했기 때문에 추가적인 실험을 하지 않았다〔16〕. 그러나 해당 기법도 이미 공개된 방법이어서 다른 도구를 이용하거나 apktool의 디스어셈블러를 수정하면 리패키징 및 앱 위변조가 가능해지므로 무결성 검증 방법의 강화는 여전히 필요하다.

VI. 뱅킹 앱의 무결성 검증 취약점에 대한 대응 방안

본 장에서는 실험을 통해서 드러난 뱅킹 앱의 무결성 검증 취약점에 대한 대응방안 및 무결성 검증 강화 방안을 구체적으로 제시한다.

6.1 실행 코드 보호

네이티브 코드에서 무결성 검증 관련 기능을 처리하기 위해서 자바 코드를 호출해야 하는 경우에는 해당 자바 코드의 보호 대책이 반드시 필요하다. 자바 코드는 역공학이 매우 쉬운 특징이 있어서 소스코드 조작을 통한 보안 기능의 우회를 가능하게 하기 때문이다.

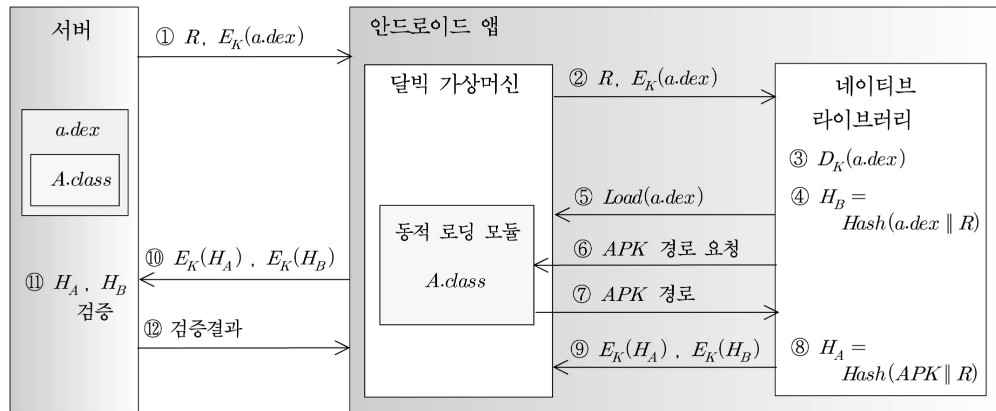
6.1.1 자바 코드의 동적 로딩

동적 로딩 기법이란, 배포된 애플리케이션에는 실행 코드를 포함시키지 않고 실행 시점에 코드를 동적으로 로딩하여 사용하는 방법을 말한다. 즉, 실행 코드의 정적 분석을 불가능하게 하며 동적 분석 또한 어렵게 함으로써 코드를 보호하는 방법이다. 안드로이드 달빅 가상머신의 DexClassLoader 클래스를 통해서 DEX 파일을 현재 실행중인 프로세스에 동적으로 로딩할 수 있도록 지원한다〔17〕.

자바 코드의 동적 로딩을 활용한 무결성 검증은 〔그림 14〕와 같이 할 수 있다. 먼저, 서버는 무결성 검증에 필요한 자바 클래스를 $a.dex$ 파일로 변환하여 저장하고 있다. 뱅킹 앱이 무결성 검증에 필요한 $a.dex$ 파일을 요청하면 서버는 세션키 K로 암호화한 $E_K(a.dex)$ 값을 난수 R과 함께 전송한다. 세션키 K는 송수신 데이터의 암호화 및 복호화를 위해서 뱅킹 앱과 서버 사이에 사용자 세션이 시작될 때 교환된 대칭키이며, 기존의 뱅킹 앱들은 모두 송수신 데이터의 암

〔표 3〕 뱅킹 앱의 무결성 검증 취약점에 대한 대응방안 및 예상효과

구분	대응방안	예상효과
실행 코드 보호	자바 코드의 동적 로딩	무결성 검증 등, 중요 자바 실행 코드의 보호
	자체 수정 코드(Self-modifying code)	자바 실행 코드의 보호 및 무결성 검증 강화
	소스코드 난독화 강화	소스코드 분석 난이도 증가
검증 대상 및 범위 확대	메모리 내용 검증	실행 메모리 검증 등 무결성 검증 강화
	무결성 검증 지점 확대	앱 위변조 공격 난이도 증가
관리적인 방안	무결성 오류 발생 계정 및 접속 IP 차단	앱 위변조 공격 시도의 사전 탐지 및 대응
	무결성 미검증 버전의 앱 사용 중지	앱 위변조 공격 시도 차단



〔그림 14〕 자바 코드 동적 로딩 기법을 이용한 안드로이드 앱 무결성 검증 예

호화를 제공하기 때문에 세션키 K가 존재한다. 뱅킹 앱의 네이티브 라이브러리는 암호화된 $a.dex$ 파일을 세션키 K로 복호화하고 해시값 H_B를 계산한 후에 달빅 가상머신에 로드한다. 해시값 H_B는 $a.dex$ 파일의 내용과 R을 조합하여 계산한다. 그런 다음, 동적으로 로드한 자바 클래스를 통해서 자바 코드 연산을 처리하고 무결성 검증용 해시값 H_A를 생성한다. 마지막으로 H_A, H_B 각각을 세션키 K로 암호화한 $E_K(H_A)$, $E_K(H_B)$값을 서버로 전송하고, 서버는 전송된 해시값을 이용하여 뱅킹 앱 및 동적 로딩 모듈인 $a.dex$ 파일의 무결성 검증을 수행한다.

이와 같은 방법을 이용하면 본 논문에서 제시한 무결성 검증 우회 기법의 방어가 가능하다. 또한, 서버에서 전송하는 동적 로딩 모듈인 DEX 파일을 수시로 변경한다면 동적 분석을 통한 공격도 효과적으로 방어할 수 있다.

6.1.2 자체 수정 코드(Self-modifying code)

코드 분석을 방지하기 위해서 실행 시점에 메모리의 실행 코드 영역에 실제 코드를 생성하는 방법이 자체 수정 코드 기법이다[18]. 즉, 안드로이드의 자바 코드에 적용된다면, 디컴파일을 통한 소스코드의 정적 분석 시에는 실제 코드를 숨기고 실행 시에 실제 코드로 치환하여 처리되기 때문에 실행 코드를 보호할 수 있다는 의미이다. 그러나 자바 코드는 디컴파일이 쉽기 때문에 자체 수정 코드 기법을 구현하더라도 실제 실행되는 코드를 포함하고 있다면 안전하지 않다. 더군다나 안드로이드 달빅 가상머신에 로딩된 바이트코드를 자바에서 직접 수정할 수 있는 방법도 없다. 따라서 안드로이드 앱에서 자체 수정 코드를 사용하기 위해서는 네이티브 라이브러리를 이용해야 한다. 별도의 네이티브 라이브러리 코드에서 리눅스의 mprotect() 함수 등을 이용하여 달빅 가상머신의 바이트코드 메모리 영역을 읽거나 쓰는 자체 수정 코드를 구현한다. 그리고 자바 코드에서 JNI 인터페이스를 이용하여 접근한다. 위와 같은 기법 적용 시에 실제 실행 코드를 안드로이드 앱에 포함시키지 않고, 서버에서 다운로드하여 메모리에 쓰는 방법을 이용하면 코드 보호 수준을 더 높일 수 있다.

또한, 자체 수정 코드 기법을 이용하여 소프트웨어의 변조 여부를 확인하는 등, 추가적인 무결성 검증 방법으로 활용할 수도 있다[19].

6.1.3 소스코드 난독화 강화

소스코드 난독화는 역공학 공격으로부터 소프트웨어를 보호하기 위한 기술이다. 본 논문에서 분석한 뱅킹 앱들의 자바 코드에는 패키지명, 클래스명, 메소드명과 필드명 등의 지시자를 의미 없는 문자로 치환하는 코드 난독화 기법을 일부 패키지에 사용했다. 그리고 일부 은행은 switch-case 문을 이용한 제어 흐름 변환 기법, 코드에서 사용하는 문자열 상수들을 암호화하는 기법들도 사용했다. 실험 과정에서, 지시자 변환 및 문자열 암호화와 같은 소스코드 난독화는 분석 시간에 많은 영향을 주었다. 따라서 자바 코드에는 문자열 암호화 등 추가적인 난독화 기법들을 적용하고 기존의 지시자 치환도 확대함으로써 소스코드 분석을 어렵게 만들어야 한다.

그리고 본 논문의 네이티브 라이브러리 분석 과정

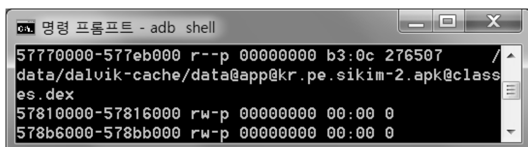
에서 확인한 바와 같이, 네이티브 코드도 어셈블리 코드 분석 및 strings 도구를 이용한 문자열 분석 공격 등이 이루어질 수 있으므로 반드시 코드 난독화를 적용해야 한다. 특히, 네이티브 코드의 기능이나 연산을 유추할 수 있는 함수명 또는 문자열들이 그대로 노출되지 않도록 난독화 기법을 적용해야 한다.

6.2 무결성 검증 대상 및 범위 확대

앱 무결성 검증 시, APK 파일에 대한 검증을 하는 것은 가장 쉬운 방법이다. 그러나 분석 및 실험결과에서 본 바와 같이, 파일 경로의 조작은 쉽게 이루어질 수 있기 때문에 검증 대상 및 범위를 확대해야 한다.

6.2.1 메모리 내용 검증

6.1.2에서 설명한 바와 같이 안드로이드의 네이티브 코드에서 달빅 가상머신에 로드된 DEX 바이트코드를 읽거나 쓰는 것이 가능하다. 안드로이드 앱이 최초로 실행될 때, 안드로이드 APK 파일 내의 DEX 바이트코드 정보인 classes.dex 파일은 APK 파일에서 추출되어 dalvik-cache 디렉토리에 저장된다. 해당 classes.dex 파일은 앱 실행을 위해서 달빅 가상머신에 적재되어야 하기 때문에 메모리에 로드된다. 〔그림 15〕는 안드로이드 특정 프로세스의 메모리 맵핑 정보를 나타내는 내용으로써, kr.pe.sikim-2.apk의 classes.dex 파일이 메모리의 0x57770000 ~ 0x577eb000 주소에 로드되어 있음을 나타낸다. 이와 같이 DEX 바이트코드 정보는 메모리에 로드되기 때문에, 네이티브 코드에서 /proc/self/maps 파일을 참조하여 classes.dex 파일이 로드된 메모리 주소를 찾아서 해당 메모리 영역을 읽으면 〔그림 17〕과 같이 실제 실행되는 바이트코드 명령들을 확인할 수 있다. 〔그림 16〕은 〔그림 17〕의 DEX 바이트코드로 변환되기 전의 실제 자바 소스코드의 일부분이다. 〔그림 16, 17〕에서 보는 바와 같이 자바 소스코드의 명령은 그대로 DEX 바이트코드로 변환되기 때문에 자바 소스코드를 위변조 하면 메모리에 로드된 바이트코드도 당연히 변경된다.



〔그림 15〕 안드로이드 /proc/self/maps 파일 내용

```
int thisPosition = 0x23115823;
Integer tempInteger = 9;
```

〔그림 16〕 자바 소스코드

```
14 04 23 58 11 23 13 05 09 00 71 10 6B 0D 05 00
0C 02 01 43 F8 10 0E 00 02 00 0A 03 1A 05 41 02
71 10 F3 0D 01 00 0C 06 71 20 85 0B 65 00 0E 00
00 00 04 00 02 00 03 00 00 00 F7 CD 05 00 0B 00
00 00 F8 10 8D 00 02 00 0C 00 15 01 07 7F F8 30
0B 00 10 03 12 10 0F 00 00 00 01 00 00 00 01 00
00 00 FF CD
```

〔그림 17〕 실제 메모리에 로드된 DEX 바이트코드

따라서 안드로이드 앱 무결성 검증 시에 APK 파일 외에 추가적으로 실행 시점의 메모리에 존재하는 DEX 바이트코드 데이터를 읽어서 서버에서 보관중인 원본 바이트코드 정보와 비교하여 위변조 여부를 검증할 수 있다. 또한 메모리의 정보는 실제 실행중인 바이트코드 명령들이기 때문에, 본 논문에서 제시한 APK 파일에 대한 무결성 검증 우회와 같은 공격을 시도하기 어렵다. 위변조 공격으로부터 방어가 필요한 중요 자바 코드에 대해서는 이와 같이 메모리의 DEX 바이트코드에 대한 무결성 검증을 추가함으로써 안전성을 강화할 수 있다.

6.2.2 무결성 검증 지점 확대

현재, 대부분의 뱅킹 앱들은 로그인 시점에 1회만 무결성 검증을 수행하고 있다. 즉, 앱을 위변조 하더라도 로그인 시점에만 무결성 검증을 우회하면 이후에는 아무런 제약 없이 사용이 가능하다. 앱 위변조 공격을 수행하는 공격자를 어렵게 만들기 위해서는 로그인 시점 뿐만 아니라 주요 기능이 처리되는 시점에 추가적으로 무결성 검증이 필요하다. 예를 들어, 계좌조회, 계좌이체, 대출 등 주요 기능이 처리될 때마다 서로 다른 부분에서 별도의 무결성 검증이 수행된다면 공격을 더욱 어렵게 만들 수 있다.

6.3 관리적인 대응방안

6.3.1 무결성 검증 오류 발생 계정 및 접속 IP 차단

앱 위변조를 통해서 무결성 검증을 우회하려는 공격자는 한 번에 성공하기 어렵기 때문에 지속적으로 오류를 발생시킬 가능성이 높다. 따라서 서버의 모니터링을 통해서 무결성 검증 오류가 지속적으로 발생하

는 계정 및 접속 IP에 대해서는 오류횟수 등의 기준을 정해서 차단해야 한다.

6.3.2 무결성 미검증 버전의 앱 사용 중지

무결성 검증이 가능하도록 앱을 업그레이드 하고 재배포 하더라도 과거에 배포된 무결성 미검증 버전의 앱이 사용 가능하면 위변조 공격의 대상이 될 가능성이 높다. 따라서 보안이 취약한 기존 배포 버전의 앱이 접속할 경우, 접속자에게 안내하여 의무적으로 신규 버전을 재설치하고 사용하도록 해야 한다.

VII. 결 론

본 논문에서는 다양한 분석 기법 및 도구들을 이용하여 안드로이드 스마트폰 뱅킹 앱의 무결성 검증 기능에 존재하는 취약점을 분석하였다. 그리고 분석 결과로부터 뱅킹 앱의 무결성 검증 기능을 쉽게 우회할 수 있는 취약점이 존재함을 확인하였다. 또한, 무결성 검증 기능을 우회하고 뱅킹 앱에 악성코드를 추가하여 재배포하는 방식으로 공인인증서 파일 및 비밀번호 등의 인증정보 유출, 계좌이체 시 수취계좌번호 변경 등 심각한 위험에 노출될 수 있음을 해킹 시나리오를 토대로 한 실험결과로 증명하였다. 실제 서비스되고 있는 국내의 주요 7개 은행의 뱅킹 앱을 대상으로 분석 및 실험을 수행함으로써 대부분의 뱅킹 앱이 취약점을 가지고 있다는 점, 그리고 그에 대한 대응방안을 구체적으로 제시하였다는 점에서 본 논문의 의의가 있다.

본 논문에서 제시한 무결성 검증 취약점에 대한 대응방안은 다음과 같다. 첫째, 자바 코드의 동적 로딩, 자체 수정 코드(Self-modifying code) 기법을 이용하여 무결성 검증과 관련된 자바 실행코드를 보호해야 하며, 자바와 네이티브 소스코드의 난독화 강화를 통해서 소스코드의 정적 분석을 어렵게 만들어야 한다. 둘째, APK 파일 외에도 달빅 가상머신에 로드된 바이트코드 메모리에 대한 무결성 검증을 추가하고 여러 실행 지점에서 무결성 검증을 실시하는 등 검증 대상 및 범위를 확대해야 한다. 셋째, 무결성 검증 오류 발생 계정 및 접속 IP의 차단 및 무결성 미검증 버전의 기존 앱 사용 중지 등 관리적인 정책을 취해야 한다. 공인인증서 비밀번호 평문 노출의 경우, 앱 무결성이 보장되면 메모리에 접근하기 어렵고 소스코드를 수정할 수 없기 때문에 탈취하기 어렵다. 금융회사 또는 금융회사에 무결성 검증 모듈을 제공하는 보안회사들은 이러한 내용들을 검토하고 대응방안을 마련함으로써 향후에 유사한 공격이 발생하지 않도록 해야 한다.

참고문헌

〔1〕 한국은행 보도자료, "2012년 3/4분기 국내 인터넷 뱅킹서비스 이용현황," 2012년 11월.

〔2〕 "Android Under Attack: Malware Levels for Google's OS Rise Threefold in Q2 2012," Kaspersky Lab, Aug. 2012, http://www.kaspersky.com/about/news/press/2012/Android_Under_Attack__Malware_Levels_for_Googles_OS_Rise_Threefold_in_Q2_2012

〔3〕 T. Bradley, "DroidDream Becomes Android Market Nightmare," PCWorld, Mar. 2011, http://www.pcworld.com/article/221247/droiddream_becomes_android_market_nightmare.html

〔4〕 전자금융감독규정, 제2012-26호, 제34조 7항, 2012년 10월.

〔5〕 최지선, 김태희, 민상식, 성재모, "스마트폰 뱅킹 앱 무결성 검증을 위한 보호기술 동향," 정보보호학회지, 23(1), pp. 54-59, 2013년 2월.

〔6〕 금융보안연구원, "모바일 앱 위변조 대응을 위한 보안기술 분석보고서," 금보원 2012-02, 2012년 7월.

〔7〕 http://code.google.com/p/android-apktool/

〔8〕 http://code.google.com/p/smali/

〔9〕 http://code.google.com/p/dex2jar/

〔10〕 http://java.decompiler.free.fr/?q=jdgui

〔11〕 "App Framework," http://developer.android.com/about/versions/index.html

〔12〕 송형주 외, 인사이드 안드로이드, 위키북스, 경기도 파주시, pp. 79-81, 2010년 10월.

〔13〕 A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google Android: A Comprehensive Security Assessment," IEEE Security and Privacy, vol. 8, no. 2, pp. 35-44, Mar.-Apr. 2010.

〔14〕 T. Bläsing, L. Batyuk, A.-D. Schmidt, S.A. Camtepe, and S. Albayrak, "An Android Application Sandbox system for

suspicious software detection," In 5th International Conference on Malicious and Unwanted Software (MALWARE'2010), pp. 55-62, Oct. 2010.
〔15〕 http://www.eclipse.org/mat/
〔16〕 T. Strazzere, "Dex Education: Practicing Safe Dex," Blackhat USA 2012, Jul. 2012, http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf
〔17〕 "Android APIs," http://developer.android.com/reference/dalvik/system/DexClassLoader.html
〔18〕 H. Cai, Z. Shao and A. Vaynberg, "Certified Self-Modifying Code," Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, vol.42, no. 6, pp. 66-77, Jun. 2007.
〔19〕 J.T. Giffin, M. Christodorescu and L. Kruger, "Strengthening software self-checksumming via self-modifying code," 21st Annual Computer Security Applications Conference, pp. 10-32, Dec. 2005.

〈저자소개〉

김 순 일 (Soonil Kim) 정회원
2003년 2월: 고려대학교 컴퓨터학과 졸업
2003년 1월~현재: 금융결제원 재직 중
2012년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 금융보안, 스마트폰 보안, 소프트웨어 보안

김 성 훈 (Sunghoon Kim) 학생회원
2006년 8월: 서울시립대학교 수학과 졸업
2009년 2월: 고려대학교 정보보호대학원 석사
2009년 1월~2011년 2월: (주)알티캐스트 CAS개발본부
2011년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 소프트웨어 보안, 소프트웨어 난독화

이 동 훈 (Dong Hoon Lee) 종신회원
1983년 8월: 고려대학교 경제학과 졸업
1987년 12월: Oklahoma University 전산학과 석사 졸업
1992년 5월: Oklahoma University 전산학과 박사 졸업
1992년 8월: 단국대학교 전자계산학과 전임강사
1993년 3월~1997년 2월: 고려대학교 전산학과 조교수
1997년 3월~2001년 2월: 고려대학교 전산학과 부교수
2001년 2월~현재: 고려대학교 정보보호대학원 교수
〈관심분야〉 암호프로토콜, 암호이론, USN 이론, 키 교환, 익명성 연구, PET 기술