

[서식 4]

※ 접수번호

H-SCOPE 결과보고서

과제유형	<input type="checkbox"/> 교과목형 <input checked="" type="checkbox"/> 기업참여형			
팀 명	20181296_홍성재			
과 제 명	안드로이드 애플리케이션 무결성 검증			
참 가 신 청				
멘토교수 (지도교수)	소속	호서대학교	지위	교수
	성명	오수현	E-mail	shoh@hoseo.edu
	전화	041 540 5716	휴대전화	041 540 5716
기업담당자 (기업멘토)	소속	주식회사 시큐센	지위	수석
	성명	심재원	E-mail	jwshim@secucen.com
	전화	010 5699 2643	휴대전화	010 5699 2643
과제기간	2023.03.01. ~ 2023.8.31.			
참 여 인 원(학생)				
성명	참여학부/학과	학번	학년	연락처
홍성재	컴퓨터정보공학부/ 정보보호트랙	20181296	4	010 7103 1477

☐ 개인정보 수집·이용 동의

구분	항 목	수집목적	보유기간
필수	학과, 학번, 학년, 성명, 전화번호, e-mail, 소속, 직위	프로그램 신청 및 운영	사업 종료 후 5년
선택	통장사본		


※ 개인정보 수집·이용에 대한 동의를 거부할 권리가 있습니다. 그러나 필수 항목의 동의를 거부할 경우 프로그램 신청에 제한을 받을 수 있으며, 선택 항목의 동의를 거부하셔도 프로그램 신청은 하실 수 있습니다.

(필수) 개인정보 수집·이용 동의	<input checked="" type="checkbox"/> 예	<input type="checkbox"/> 아니요
(선택) 개인정보 수집·이용 동의	<input checked="" type="checkbox"/> 예	<input type="checkbox"/> 아니요

H-SCOPE에 대한 결과보고서를 위와 같이 제출합니다.

2023년 00월 00일

지도 교수 : 오수현 (인)

대표 학생 : 홍성재 

AI·SW중심대학사업단장 귀하

요 약 서

1. 주요 목표

- 애플리케이션의 디컴파일, 리컴파일(느리패키징), 리사이닝 등의 과정을 이해하고 해당 과정 중간에 Smali 코드 주입을 하여 앱을 임의로 변조시켜 서버를 통해 무결성을 검사하게 하였다.

2. 주요 결과물

- 애플리케이션을 디컴파일, 리컴파일, 리사이닝 하며 중간에 설정해둔 경로로 Smali 코드와 Androidmanifest.xml, 라이브러리들을 자동으로 주입하는 실행 파일
- 목표로 삼은 애플리케이션을 대상으로 만든 Smali 코드, 라이브러리, AndroidManifest.xml 파일들
- 약속된 알고리즘과 프로토콜을 통해 암호통신을 하며 설정해둔 SHA1 해시값을 통해 접속한 대상의 애플리케이션의 무결성을 체크하는 리눅스 기반 서버 실행 파일
- 기존 애플리케이션에 작성한 Smali 코드와 AndroidManifest.xml, 라이브러리를 주입하여 최초 실행 시 내 서버에 접속하여 애플리케이션의 서명 데이터를 확인하고 무결성을 검사하여 앱의 위변조 유무를 판단 후 정상일 경우 기존 애플리케이션이 실행되고, 비정상일 경우 앱이 종료된다.

3. 결과물 활용방안

- 애플리케이션의 서명 데이터를 검증하기에 마지막으로 서명한 인증서만 알고 있다면 해당 애플리케이션이 배포 이후 디컴파일 되었는지 아닌지를 알 수 있고, 기타 다른 애플리케이션에도 도입할 수 있다면 어느 앱이든 서명 검증을 통해 위변조 사실을 판단 할 수 있을 것이다.

목 차

1. 안드로이드 애플리케이션

1-1. 애플리케이션 구성	4
1-2. smali 코드	4
1-3. 애플리케이션의 디컴파일, 리패키징, 리사이닝 과정	5
1-4. 애플리케이션의 서명	5

2. hSolution

2-1. hSolution의 개발 이유	8
2-2. 동작 흐름	8
2-3. 동작 실행 및 결과	10
2-4. 개선해야할 부분	19
2-5. 한계점	21
2-6. 오류코드	22

3. 참고문헌

3-1. 참고문헌	23
-----------------	----

1. 안드로이드 애플리케이션

1-1. 애플리케이션의 구성

apk는 안드로이드 애플리케이션의 확장자이며, 이는 압축 파일이다. 따라서 압축 프로그램으로 쉽게 압축 해제가 가능한데 apk 파일을 압축 해제해보면 크게 다음과 같이 나뉜다. META-INF, assets, res, lib(혹은 libs)폴더와 AndroidManifest.xml, class.dex 파일로 나뉜다. 다음의 표로 간단히 정리해보았다.

이름	폴더 / 파일	설명
META-INF	폴더	인증 서명 정보 등의 데이터
assets	폴더	앱의 자원 (용량이 크다.)
res	폴더	앱의 자원 (용량이 작다.)
lib	폴더	앱의 라이브러리
AndroidManifest.xml	파일	앱의 정보 파일
class.dex	파일	Class 파일을 Dalvik이 인식할 수 있도록 변환한 파일

이 중 class.dex 파일은 기계어의 바이트 코드로 되어있으며, 실행 시 궁극적으로 실행되는 코드가 포함되어있다. 단순히 에디터와 같은 프로그램으로 열어도 볼 수 없기에 컴파일러를 사용하여 디컴파일 한 것이 smali 코드이다.

1-2. SMALI 코드

Dex 바이트 코드를 디컴파일 한 smali 코드는 dex 바이너리를 사람이 읽을 수 있도록 표현한 코드이다. 어셈블리로 되어있으며 이는 수정이 가능하다.

이번 프로젝트에선 기존의 코드를 수정하는 것이 아닌 새로운 클래스를 위해 새로운 smali 코드를 작성하였고, 기존 애플리케이션에서 Application 클래스를 사용하는 클래스가 없었기 때문에 주입할 클래스를 Application 클래스를 상속받아 메인 액티비티 보다 우선 작동하게 만들 수 있었다.

다음은 smali 코드의 예시이다.

```
# direct methods
.method public constructor <init>()V
    .registers 2

    .line 24
    invoke-direct {p0}, Landroid/app/Application;-><init>()V

    const/4 v0, 0x0

    .line 25
    iput v0, p0, Lcom/tpcstld/twozerogame/hSolution;->ret:I

    return-void
.end method
```

1-3. 애플리케이션의 디컴파일, 리패키징, 리사이닝 과정

애플리케이션의 디컴파일이란, 리버스 엔지니어링의 과정이며, 애플리케이션을 사람이 읽을 수 있는 프로그래밍 언어로 변환하는 것이다. 이번 프로젝트의 경우 안드로이드 환경에서 진행했기 때문에 목표의 애플리케이션을 디컴파일하여 원본의 java 코드를 획득 할 수 있었다.

리패키징은 디컴파일의 반대인 리컴파일의 과정이며 프로그래밍 언어에서 실행 가능한 애플리케이션으로의 변환이며 변환 후, 리사이닝 과정을 거쳐 변환된 애플리케이션에 서명 과정을 진행하여 최종적으로 설치 및 실행이 가능한 애플리케이션을 획득 할 수 있다.

이번 프로젝트에서 중요한 부분은 리패키징 이후 리사이닝 과정이 진행되므로, 코드의 모든 개발 및 수정이 이루어진 뒤 서명 과정이 진행된다. 다시 말해, 서명이 다시 된 경우는 해당 애플리케이션의 수정이 이루어졌다고 볼 수 있으며, 이는 무결성이 깨진 것이다.

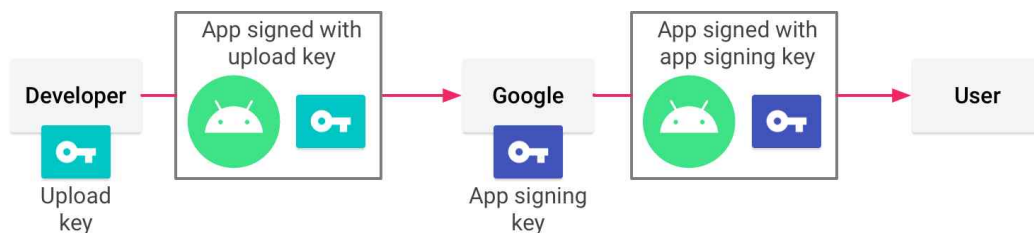
서명의 경우 키스토어를 통한 서명 과정이 진행되는데 키스토어를 통한 서명이 진행 될 때, 키스토어에 대한 비밀번호가 필요하므로 악의적인 목적의 사용자가 앱을 디컴파일, 리패키징, 리사이닝을 하는 경우 디컴파일 이전 사용한 키스토어의 비밀번호를 모른다는 가정 하에 이전과 다른 자신의 키스토어로 서명이 진행되므로 원본의 서명 데이터와 달라질 수밖에 없다.

1-4. 애플리케이션의 서명

애플리케이션의 서명이란, 애플리케이션이 배포되기 위해 반드시 진행되어야 하는 절차

이다. 앱을 개발하는 개발자가 배포 전 서명을 통해 개발자가 배포하는 앱을 증명하기도 하며, 앱이 업데이트될 때 서명 데이터를 통해 같은 개발자, 배포처에서 부터 진행되는 업데이트임을 증명하기도 한다. 구글 플레이스토어에 앱을 배포하는 경우 다음과 같은 서명 과정이 진행된다.

개발자가 개발한 앱을 개발자의 키로 서명하여 구글에 배포를 요청한다. 구글은 앱과 앱의 서명 데이터를 비교하여 요청한 개발자가 개발한 앱인지 비교하고 일치하는 경우, 구글의 키로 다시 서명하여 플레이스토어에 올려 배포한다. 이후 사용자가 구글 플레이스토어에서 앱을 설치하는 경우, 해당 앱이 구글에서 서명되었는지 확인하고 일치하는 경우 앱이 설치된다.



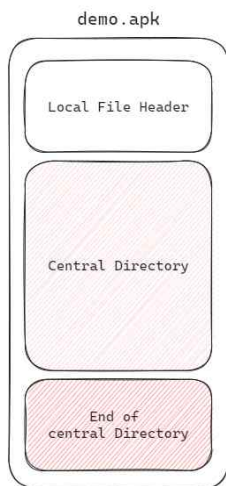
[그림 1. 애플리케이션 서명]

이는 전자서명의 부인방지 성격이 드러나는 부분이다. 구글은 최초 개발자로부터 개발자가 맞는지 확인하고, 유저는 구글이 배포하는지 확인하는 데 사용된다.

앱 서명은 v1부터 v4까지의 서명 체계가 존재하며, 이번 프로젝트에선 v2의 서명 체계를 사용했다. v1의 경우 JAR Signer를 통해 서명이 가능하며, v1의 특징으로 앱을 압축 해제한 뒤 모든 파일을 청크 단위로 쪼개어 해시하고 모든 해시들을 종합하여 최종 해시값을 통해 서명한다.

따라서 서명을 검증하기 위해선 앱의 전체를 압축 해제해야 하며 이는 속도가 매우 느리다. 또한 최종 앱에 대해 서명이 됐더라도 기존 파일 이외의 파일이나 코드가 추가되더라도 감지하지 못하는 것이 특징이다. 왜냐하면 최종 앱의 해시가 아닌 압축 해제 했을 때 모든 파일의 머클 트리와 유사한 해시데이터를 사용하기 때문에 추가되는 데이터에 대해 대응이 불가하다. 따라서 안드로이드 7.0 이후부터 v2 서명을 권장한다.

v2 서명 체계는 JAR Signer가 아닌 APK Signer를 사용하며, 앱의 패키지 자체를 인증한다. 따라서 기존 v1과는 달리 최종 앱에 대한 인증을 지원하므로 중간에 데이터가 추가된다면 서명이 무효화 되며 무결성을 해치게 된다. 이번 프로젝트에선 v2 서명체계를 사용했으며, v2의 경우 파일의 정렬을 도와주는 zipalign이 우선 실행 된 뒤 진행되어야 한다. zipalign이 실행되는 경우 앱의 구조 중 Local File Header에 해당하는 부분의 필드 크기가 변하기 때문이다.



[Signature v1]

- jarsigner 사용 (jarsigner → zipalign)
 - jarsigner는 apk와 aab 파일 모두 사용 가능
- v1서명은 메타데이터와 같은 apk의 일부 구성요소를 보호하지 않는다.
- jarsigner 는 META-INF/MANIFEST.MF에 나열된 항목이 포함된다.
- 기존 데이터 이외에 새로 추가 된 데이터에 대해 대처 불가
- 검증을 위해선 모든 entry를 압축 해제해야 한다.

[Signature v2]

- apksigner 사용 (zipalign → apksigner)
 - apksigner는 apk 파일에만 사용 가능
 - Google에서 apk 서명 시 apksigner 권장
- Android 7.0 (Nougat) 이후 사용
- 앱 패키지를 구성하는 모든 파일을 개별로 인증하지 않고 패키지 자체를 인증
- ∴ 서명 후 zipalign 실행 불가
 - apksigner → zipalign 수행 시 서명이 무효화

[zipalign]

- 보관 파일 중 압축되지 않은 모든 파일이
파일 시작 부분을 기준으로 정렬되도록 하는 ZIP 파일 정렬 도구
- 정렬을 위해 Local File Header 섹션의 "extra" 필드 크기를 변경한다

[그림 2. Signature v1 & v2 요약]

2. hSolution

※ 이번 프로젝트에서 작성 된 코드, 또는 환경의 경우 타겟으로 지정한 애플리케이션에서 가능한 것이지만 다른 애플리케이션에선 수정이 필요하다. ※

- | | | |
|--------------|---|---------------------------------------------------------------------------------------------------------------------------|
| ♦ os | : | Linux(WSL v1.2.5.0), Window 11 |
| ♦ test phone | : | Pixel XL (API : 33, Android 13.0, ABI : x86_64, by Android Studio VM) |
| ♦ target | : | 2048 by Jerry Jiang (2048, tpcstld) |
| ♦ tools | : | apktool (v2.4.1)
jadx-gui (v1.4.6, Java VM : OpenJDK 64-bits, v19.0.2)
vscode (v1.78.2)
Android Studio(v11.0.15) |
| ♦ lang | : | C, Java, Smali |
| ♦ crypto | : | Symm : ARIA128 / CBC / PKCS5
Asym : RSA / SHA256
(SECUCEN의 EdgeCrypto 라이브러리를 지원받아 사용했음) |

2-1. hSolution의 개발 이유

22년도 동계 계절 학기부터 23년도 1학기까지 현장실습을 이어오며 여러 암호 알고리즘을 다루어왔고 또 직접 만들어보기도 했다. 알고 있던 지식부터 모르는 지식까지 차근차근 배워나가며 실습을 하던 중, 안드로이드 애플리케이션의 Smali Code Injection에 대해 공부할 기회가 생겼다. 아주 단순하게 내부 코드를 수정해서 게임 점수를 조작하거나, 코드를 더 추가해서 다른 애플리케이션을 호출하거나 하는 등의 동작이 가능했다. 물론 타겟이 됐던 애플리케이션은 난독화가 되어있지 않고, 네트워크보단 로컬에서 동작하는 애플리케이션을 타겟으로 삼았다.

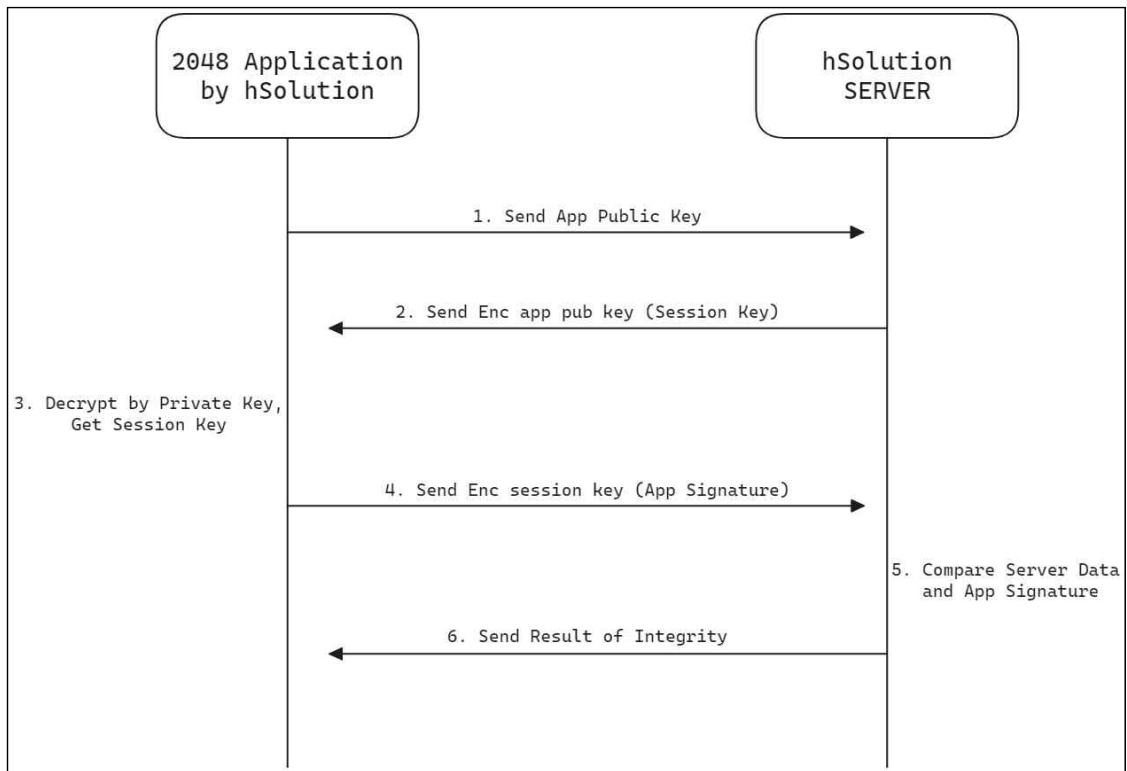
이후 이번 프로젝트의 소식을 듣고 주식회사 시큐센의 앱 아이언에 대해 들었다. 내가 진행했던 Smali Code Injection에 대해 방어하는 솔루션이었다. 비슷하게나마 한번 만들어보고 싶었다. 내가 할 수 있는 한계점을 명확히 하고, 하고자 하는 목표를 정확히 하여 의도하는 동작이 수행되게 만들었다.

2-2. 동작 흐름

최초 개발자가 앱을 개발한다. 이후 개발자는 의뢰를 통해 앱 파일과 최종 배포에 사용될 서명의 지문을 hSolution 측에게 알려준다. hSolution은 앱 아이디와 해당 앱의 서명 데이터를 서버에 등록하고, 앱 파일은 디컴파일을 진행하고 smali 코드를 통해 서버로 접속하게 될 코드를 주입한다. 이후 리패키징, 리사이닝 과정을 거친 뒤 다시 개발자에게 앱을 보낸다.

hSolution을 거친 앱은 다음과 같은 과정이 추가된다. Application 클래스를 상속받아 메인 액티비티 이전에 실행되며, 최초 실행 시 hSolution의 서버에 접속하여 실행중인 서명 데이터를 전송하고 서버로부터 응답을 받는다. 응답에 따라 무결성을 검증하고, 검증에 실패 시 앱이 중지되며, 반대로 검증에 성공 시 앱이 정상 작동한다. 검증에 사용되는 서명 데이터는 개발자가 hSolution에게 알려준 최종 배포용 서명 데이터이다.

서버와의 통신은 간단한 암호 통신이 이루어진다. 앱이 서버에 접속 시, 서버에 자신의 공개키를 보내고 서버는 해당 공개키를 바탕으로 세션키를 암호화 한다. 앱은 서버로부터 Enc_{앱 공개키}(세션키)를 받고 이를 앱의 개인키로 복호화 하여 세션 키를 획득한다. 이후론 획득한 세션 키로 통신이 이뤄진다.

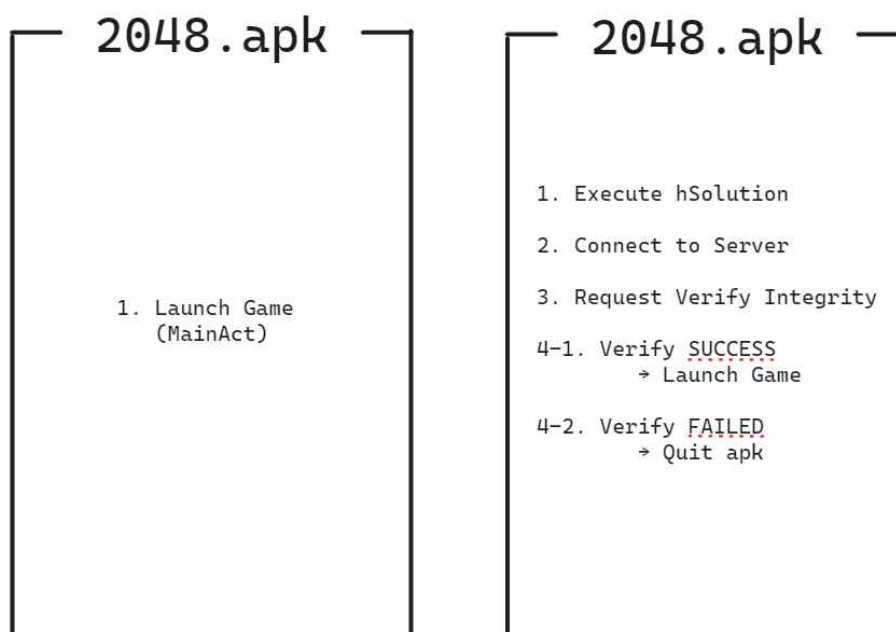


[그림6. 애플리케이션과 서버의 통신 과정]

따라서, 개발자는 hSolution을 거친 앱을 배포용 키로 서명하여 배포하고, 유저는 해당 앱을 설치 후 실행할 때 무결성을 확인할 수 있다. 또한 이 검증과정을 포함한 hSolution의 동작은 스레드 환경에서 진행되기 때문에 앱이 블록 상태에 걸리지 않는다.

[BEFORE]

[AFTER]



[그림3. hSolution 동작 흐름]

검증에 있어 서명 데이터를 사용한 이유는 두 가지 이다. 첫째는 서명 v2부터 모든 개발이 끝난 이후 서명이 들어가고, 그 이후 앱 수정이 이루어지면 서명이 깨지며 무결성이 무효화되는 특징이 있어 사용했다. 애플리케이션이 실행 중에 데이터를 가져올 수 있으며, 서명의 지문 값을 사용하기에 유일한 데이터를 가져올 수 있다는 장점이 있었다. 때문에 개발을 마친 앱의 무결성을 확인할 수 있고, 값의 추출이 자유로웠기 때문에 사용하기에 용이했다. 둘째로, 구글 플레이 스토어에 앱을 등록할 때 재서명이 이루어지면서 애플리케이션의 데이터가 변할 수 있기 때문이다. 안드로이드의 구글 플레이 스토어의 경우, 앱을 업로드 하면서 앱 내부에 구글 API가 추가되기도 하고, 구글에서 관리하는 업로드용 서명 키스토어가 따로 관리되기 때문이다. 이 과정은 정확히 개발자가 어느 부분에서 변경되는지 알기 힘들고 또한 개발을 마친 애플리케이션의 데이터가 변할 수 있기 때문이다. 처음 이 솔루션을 구상할 땐, 무결성이라는 성격을 생각해서 애플리케이션 apk 파일에 대한 해시값을 최초에 등록하고 사용하려했는데, 자문을 구하고 여러 문서를 읽어 본 결과 구글 플레이스토어에서 위와 같은 과정이 추가될 수 있기 때문에 사용하기 힘들다고 판단했다.

파일의 해시값의 경우, apk 파일이 사용자 스마트폰에 저장되는 게 아닌 설치되기 때문에 원본 파일의 해시값을 그 때마다 구하기 힘들고, 그렇다고 라이브러리 내부에 해시값을 넣기엔 모든 애플리케이션마다 각각의 라이브러리가 제공되어야하기 때문에 유지보수에서 매우 힘들어지겠다고 생각했다.

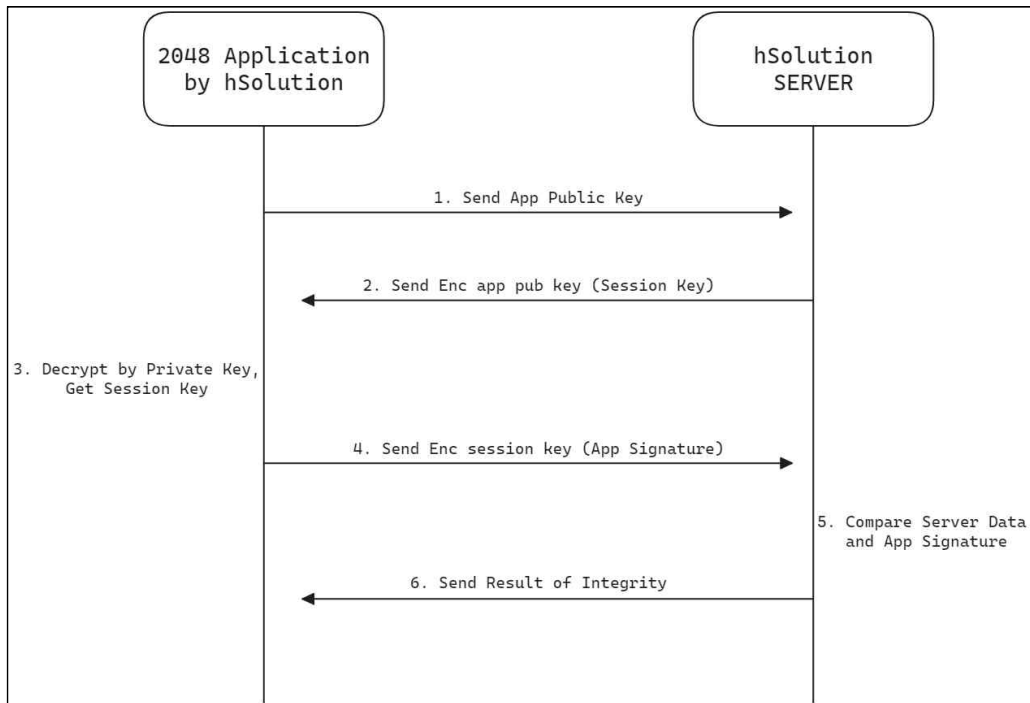
2-3. 동작 실행 및 결과

전체 수행 과정

- | | |
|---------------------------------------------|-------------------|
| 1. ADB를 통한 APK 추출 | (Window) |
| 2. 추출한 APK 디컴파일 & 정적 분석 | (Linux) |
| 3. 분석을 토대로 AndroidManifest.xml과 Smali 코드 작성 | (Window, Linux) |
| 4. 작성한 파일 주입 | (Linux) |
| 5. 리컴파일 | (Linux) |
| 6. 리사이닝 | (Linux) |
| 7. 앱 구동 확인 (동적 분석) | (Android, Window) |

윈도우 환경에서 ADB로 해당 애플리케이션을 추출하고 jadx-gui 프로그램을 통해 내부를 보면, 최초 실행 시 com.tpcstld.twozerogame 패키지의 MainActivity가 실행된다.

내부 코드	구동 화면
 <p>[그림4. 원본 애플리케이션의 최초 진입점]</p>	 <p>[그림5. 원본 애플리케이션 구동 화면]</p>



[그림6. 애플리케이션과 서버의 통신 과정]

위와 같은 기존 애플리케이션에 hSolution을 추가하기 위해 구성한 Native Library의 헤더는 다음과 같다.

- ◆ `private native int InitCrypto (String sighPath);`
 - 제공받은 EdgeCrypto를 안드로이드에서 사용하기 위한 서명 검증 함수
 - ABI별 존재하는 EdgeCrypto.sign 파일의 절대 경로를 넘겨줘야한다.
 - 애플리케이션 apk 파일 내부에 존재하는 서명 데이터를 원본 그대로 사용하기 위해 서명 데이터를 로컬 디렉토리에 복사한 뒤 로컬 디렉토리의 절대 경로를 넘겨줘야한다.
 - 성공 시 0을 return, 실패 시 해당하는 오류 코드 return
- ◆ `private native int verifyIntegrity (String signData);`
 - 내부적으로 서버에 접속할 소켓을 생성 및 연결한다.
 - 연결 이후엔 간단한 키교환을 통해 세션키로 대칭키 암호 통신을 한다.
 - 현재 실행중인 앱의 서명 데이터를 서버에 보내어 무결성을 체크한다.
 - 무결성이 지켜졌을 경우 0을, 이외엔 0이 아닌 값을 return

hSolution.smali 코드는 assets/ 폴더에 존재하는 파일을 .jar 파일로 압축하여 내부 로컬 디렉토리로 복사한다. 그 뒤 로컬에서 .jar파일을 압축해제 한 뒤 해당 로컬 디렉토리의 절대 경로를 IninCrypto 함수로 전달해준다. 정상적으로 EdgeCrypto의 서명 검증이 이뤄졌으면 verifyIntegrity를 통해 무결성을 검사하고 이후 앱을 정상 실행할 것인지 종료할 것인지 판단한다. 다음은 smali 코드로 주입되는 hSolution의 간단한 코드와 도식화한 통신 과정이다. assets/ 폴더에서 직접 로드하지 않고 압축한 jar 파일을 내부 로컬에 복사해 로드하는 이유는 뒤에 기술되어있다.

```

import android.app.Application
...

public class hSolution extends Application {
    public void onCreate( ) {
        try {
            // apk 파일 내부 assets 폴더에 접근해 라이브러리와 서명 파일들의 압축 jar 파일을
            // 내부 로컬에 복사한 뒤 압축을 해제하고 라이브러리를 로드하는 함수
            nativeLibLoader(jarFilePath, localDir);
        } catch (IOException e) {
            e.printStackTrace( );
            exitApp( );
        }
        // 서명 검증이 됐는지 판단
        ret = InitCrypto((localDir + "/libEdgeCrypto.sign").toString( ));
        if (ret != 0) ...

        // 무결성 검증이 됐는지 판단
        ret = verifyIntegrity(getApkSignature(getApplicationContext( )));
        if (ret != 0) ...
    }
}

```

라이브러리를 로드하고 서버에 접속해 검증을 요청하고 그 결과를 받아오는데 까지 약 1.408초가 걸린다. 게다가 스레드 환경에서 돌아가기 때문에 메인 액티비티에 끼치는 영향은 굉장히 적을 것 이라고 생각한다.

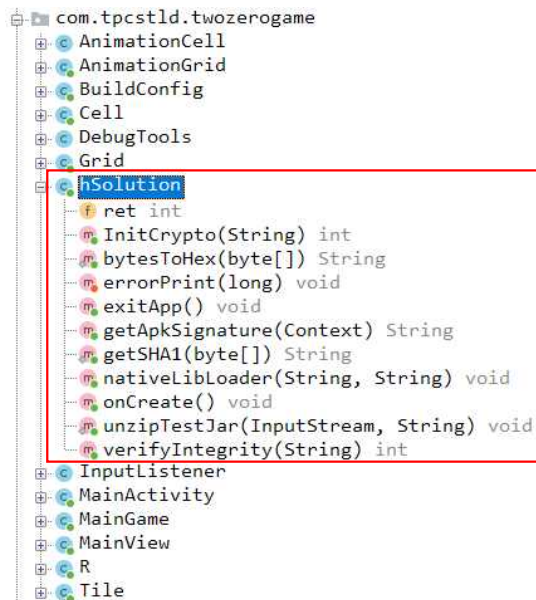
다음은 검증에 성공했을 경우 생기는 로그(logcat 기반)이다.

```

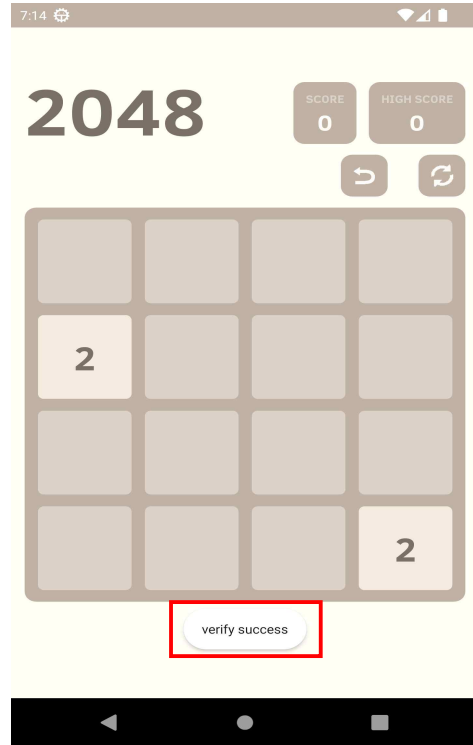
----- beginning of main
----- beginning of system
hSolution      com.tpcstld.twozerogame      D  loaded native lib
hSolution_native_lib  com.tpcstld.twozerogame      D  success init
hSolution      com.tpcstld.twozerogame      D  func:initCrypto() success
hSolution_native_lib  com.tpcstld.twozerogame      D  make key pair
hSolution_native_lib  com.tpcstld.twozerogame      D  make socket
hSolution_native_lib  com.tpcstld.twozerogame      D  [Phone] is connected
hSolution_native_lib  com.tpcstld.twozerogame      D  recv from server enc symm key
hSolution_native_lib  com.tpcstld.twozerogame      D  get symm key
hSolution_native_lib  com.tpcstld.twozerogame      D  encrypt data
hSolution_native_lib  com.tpcstld.twozerogame      D  success send enc data
hSolution      com.tpcstld.twozerogame      D  verify success

```

hSolution 로그는 Java에서 생기는 로그이고, hSolution_native_lib 로그는 Native Library에서 생기는 로그이다. 위에서 설명한 것과 동일한 과정으로 진행이 됐음을 볼 수 있다. 마지막의 로그인 “verify success”의 로그는 서버에서 로컬로 보낸 메시지이며, 로컬에서 서버로 전송한 서명 지문 데이터가 서버의 데이터와 일치하는 경우에만 보낸다.



[그림 7. Smali 코드의 주입 결과 내부 코드]



[그림 8. 검증 성공 시 토스트 메시지]

주입한 코드에 따라 com.tpcstld.twozerogame 패키지에 hSolution 클래스가 추가되었고, 이 클래스는 Application을 상속받기 때문에 MainActivity의 onCreate 보다 hSolution의 onCreate가 우선 실행된다. hSolution의 smali 코드는 jadx-gui 환경에서 Java 언어를 Smali 코드로 변환해주기 때문에 jadx-gui의 도움을 받아 작성하였다.

다음은 검증 과정이 실패했을 경우이다. 검증이 실패할 경우는 다음과 같다. 첫째로, Native Library의 로드가 실패했을 경우이다. Native Library는 기본 assets/ 폴더 내의 있는 파일들을 스마트폰 내부 로컬 디렉토리에 복사한 뒤, 내부 디렉토리의 절대 경로를 따라 로드된다. 파일을 다 읽지 못했거나, 복사의 일부가 실패하거나 하는 경우 오류가 발생 할 수 있다. 해당 오류는 try catch 문으로 처리했으며, 발생 시 로그를 찍고 앱을 종료하게 된다.

둘째로, 암호 모듈 EdgeCrypto의 서명 검증이 실패했을 경우이다. 해당 암호 모듈은 CMVP를 인증받은 모듈로, CMVP 초기화 과정에서 서명 검증이 필요하다. 안드로이드

환경에서 서명 검증을 위해 위에서 설명한 라이브러리를 로드하는 과정에 서명 파일이 같이 존재하고, 해당 서명 파일 또한 로컬에 저장된다. 또한, assets/ 폴더에서 로드하는 경우 서명 파일의 무결성이 깨질 수 있기 때문에 로컬에 복사하여 절대경로로 서명 검증을 수행한다. 서명 검증의 경우, 생길 수 있는 오류 상황은 입력받은 해당 경로에 서명 파일이 존재하지 않거나, 권한의 문제로 해당 경로를 읽지 못하는 경우, 마지막으로 해당 서명 파일로 서명 검증이 실패한 경우이다. 각각의 오류 상황은 EdgeCrypto의 오류 헤더 내의 정리되어있고, 만약 서명 검증이 실패한 경우, 실패의 원인에 대한 오류 코드가 반환된다.

마지막으로, 무결성 검증 과정의 오류이다. 위의 두 과정이 정상적으로 실행 된 후, 검증을 시작한다. 검증 과정은 디컴파일시 노출되는 것을 방지하기 위해 Native Library에서 구현되어있다. 본 함수는 실행중인 해당 앱의 서명 데이터가 입력으로 들어간다. 최초 실행 시, 해당 스마트폰의 비대칭 키를 생성하고, 소켓을 연결 한 뒤, 위에서 설명한 검증 과정이 실시된다. 이 모든 과정에서 생길 수 있는 오류들은 다음과 같다. 비대칭 키 생성 실패의 경우, 소켓 생성 실패의 경우, 소켓 연결의 실패의 경우, 소켓을 통한 수신, 발신의 실패의 경우, 대칭·비대칭 암호·복호화의 경우이다. 이 무결성 검증 과정의 경우 생길 수 있는 오류 상황이 많고, 기존 시스템 함수를 사용한 경우도 많아 다음과 같이 표로 정리해보았다.

오류 발생 경우	해당 함수 출처
비대칭 키 생성 오류	EdgeCrypto
소켓 생성 오류	System
소켓 연결 오류	System
소켓 수신, 발신의 오류	System
대칭·비대칭 암호·복호화의 오류	EdgeCrypto

발생하는 오류에 대해 처리하기 위해 try catch문이 아닌, 정수형 ret 변수를 통해 ret이 0인지 아닌지 판단을 통해 처리하였다.

Native Library의 로드 실패와, 암호 모듈 서명 검증의 실패의 경우 로그를 찍고 앱이 종료되는 반면, 이 두 과정이 정상 동작 한 뒤 무결성 검증의 과정에서 오류가 발생했을 경우, 앱 위에 토스트 메시지로 “verify failed” 라는 문구가 뜨고 앱이 종료되게 처리하였다.

다음은 라이브러리 로드와 암호 모듈 서명 검증의 성공 이후 앱 무결성 검증에 실패한 경우의 로그이다.


```

----- beginning of main
----- beginning of system
hSolution          com.tpcstld.twozerogame      D  loaded native lib
hSolution_native_lib com.tpcstld.twozerogame      D  success init
hSolution          com.tpcstld.twozerogame      D  func:initCrypto() success
hSolution_native_lib com.tpcstld.twozerogame      D  make key pair
hSolution_native_lib com.tpcstld.twozerogame      D  make socket
hSolution_native_lib com.tpcstld.twozerogame      D  [Phone] is connected
hSolution_native_lib com.tpcstld.twozerogame      D  recv from server enc symm key
hSolution_native_lib com.tpcstld.twozerogame      D  get symm key
hSolution_native_lib com.tpcstld.twozerogame      D  encrypt data
hSolution_native_lib com.tpcstld.twozerogame      D  success send enc data
hSolution          com.tpcstld.twozerogame      D  verify failed

```

마지막으로 애플리케이션 원본으로부터 hSolution을 진행하는 총 과정의 리눅스 환경 실행 화면이다. 해당 실행 파일은 makefile로 빌드했으며 내부에서 apktool과 apksigner를 사용하였다. 실행하는 모든 단계에서 오류 발생 시 오류코드를 처리하게 설계되었고 해당 실행 파일이 종료된다. system() 함수를 통해 시스템 명령어를 호출하였고 해당 함수의 반환값을 정수형 변수 ret 에 담아 처리하였다.

1. hSolution 실행 전

```

.target_apk
├── h_server.keystore    ← 리사이닝의 필요한 키스토어
└── origin_2048.apk     ← 원본 APK

```

2. 애플리케이션 원본 디컴파일 과정

```

===== [ START DECOMPILE APP ] =====
I: Using Apktool 2.4.1 on origin_2048.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/user/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...

```

3. 디컴파일된 경로에 코드 및 라이브러리 주입

```
===== [ START rm / cp FILES ] =====
--> rm Androidmanifest.xml
--> cp Androidmanifest.xml
--> inject smali code
added manifest
adding: inject_lib/assets/(in = 0) (out= 0)(stored 0%)
adding: inject_lib/assets/arm64-v8a/(in = 0) (out= 0)(stored 0%)
adding: inject_lib/assets/arm64-v8a/libEdgeCrypto.sign(in = 260) (out= 265)(deflated -1%)
adding: inject_lib/assets/arm64-v8a/libEdgeCrypto.so(in = 358736) (out= 170275)(deflated 52%)
adding: inject_lib/assets/arm64-v8a/libhSolution.so(in = 9488) (out= 4016)(deflated 57%)
adding: inject_lib/assets/armeabi-v7a/(in = 0) (out= 0)(stored 0%)
adding: inject_lib/assets/armeabi-v7a/libEdgeCrypto.sign(in = 260) (out= 265)(deflated -1%)
adding: inject_lib/assets/armeabi-v7a/libEdgeCrypto.so(in = 264104) (out= 147604)(deflated 44%)
adding: inject_lib/assets/armeabi-v7a/libhSolution.so(in = 6640) (out= 3421)(deflated 48%)
adding: inject_lib/assets/x86/(in = 0) (out= 0)(stored 0%)
adding: inject_lib/assets/x86/libEdgeCrypto.sign(in = 260) (out= 265)(deflated -1%)
adding: inject_lib/assets/x86/libEdgeCrypto.so(in = 386920) (out= 173920)(deflated 55%)
adding: inject_lib/assets/x86/libhSolution.so(in = 9408) (out= 3977)(deflated 57%)
adding: inject_lib/assets/x86_64/(in = 0) (out= 0)(stored 0%)
adding: inject_lib/assets/x86_64/libEdgeCrypto.sign(in = 260) (out= 265)(deflated -1%)
adding: inject_lib/assets/x86_64/libEdgeCrypto.so(in = 371304) (out= 170971)(deflated 53%)
adding: inject_lib/assets/x86_64/libhSolution.so(in = 10272) (out= 3961)(deflated 61%)
--> zip jar assets
--> make dir
--> cp assets
--> mv jar file
```

4. 주입 이후 다시 애플리케이션으로 만드는 리컴파일

```
===== [ START RECOMPILE APP ] =====
I: Using Apktool 2.4.1
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs... (/lib)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
```

5. 리컴파일 이후 앱 리사이닝

```

===== [ START RESIGNING APP ] =====
Keystore password for signer #1:           // insert keystore passwd
H_solution success

```

6. 최종 결과물

```

.target_apk
|—— h_server.keystore      ← 리사이닝의 필요한 키스토어
|—— origin_2048.apk        ← 원본 APK
|—— de_2048/               ← 디컴파일 된 디렉토리
|—— re_2048.apk            ← de_2048/ 폴더를 리컴파일한 결과
|—— re_2048_signed.apk     ← re_2048.apk을 리사이닝한 최종 결과

```

2-4. 개선해야할 부분

가장 먼저 개선해야 할 부분은, 바로 위에서 설명한 오류 처리 과정이다. 해당 오류 처리를 Java에서 처리를 하지만, 라이브러리를 로드하는 과정 외엔 정수형 변수를 통해 오류코드를 판단하여 처리한다. 이는 Java스럽지 못하다고 생각한다. 내 임의대로 Exception을 정의하고 이후 과정도 try catch 문으로 처리할 수 있었지만, 가능한 앱의 지장을 적게 하고 싶어 hSolution이라는 클래스 하나로 처리하고 싶었다. 또한 라이브러리 로드, 암호 모듈 서명 검증 이후 무결성 검증 과정에서 오류 발생으로 앱이 종료되었을 때, 토스트 메시지를 띄우지만, 앱 종료로 인해 사용자가 해당 토스트 메시지를 보기 전에 앱이 종료 된다. 해당 앱 사용자에게 친화적으로 만들려면 해당 오류 발생 사실을 알리는 게 맞다고 생각한다. 토스트 메시지로 처리하기 보단, 오류 발생 사실과, 확인 버튼을 띄워 해당 버튼이 눌렸을 경우 앱이 정상 종료되게 처리하는 게 더 낫다고 생각한다. 토스트 메시지로 처리한 이유는 아무 스레드에서 호출이 가능하며, Activity가 아닌 구성 요소에서도 제약 없이 사용 가능하기 때문이다. 하지만, hSolution의 전 과정이 스레드에서 처리가 된다. hSolution의 경우 Application의 상속을 받기 때문에 전역에 영향이 가는데 해당 환경에서 스레드로 처리할 경우 해당 스레드가 전역에 영향을 미칠 수 있기 때문에 동기화 문제가 생길 수 있다. 이번 프로젝트 타겟의 경우 무겁거나 복잡한 애플리케이션이 아닌 간단한 애플리케이션 이였기 때문에 큰 문제없이 수행하였지만, 후에 더 크고 복잡한 애플리케이션을 다룰 경우 mutex 등과 같이 동기화에 유의해야 할 부분이다. 만약 앱 종료 부분도 디컴파일 방지를 위해 네이티브에서 작동하는 것이 바람직하다면 다음과 같은 코드로 네이티브 (C언어) 에서도 앱을 종료할 수 있다.

```

JNIEXPORT void JNICALL Java_com_example_MyClass_runFinalization(JNIEnv *env, jobject obj) {
    // Java의 System.runFinalization() 실행
    jclass cls = (*env)->FindClass(env, "java/lang/System");
    jmethodID mid = (*env)->GetStaticMethodID(env, cls, "runFinalization", "()V");
    (*env)->CallStaticVoidMethod(env, cls, mid);
}

JNIEXPORT void JNICALL Java_com_example_MyClass_exit(JNIEnv *env, jobject obj) {
    // Java의 System.exit(1) 실행
    jclass cls = (*env)->FindClass(env, "java/lang/System");
    jmethodID mid = (*env)->GetStaticMethodID(env, cls, "exit", "(I)V");
    (*env)->CallStaticVoidMethod(env, cls, mid, 1);
}

```

다음으로 개선이 필요한 부분은 소켓 통신의 프로토콜이다. 이번 프로토콜의 경우는 아주 간단하게 구현한 프로토콜을 도입하였다. 해당 프로토콜은 출처의 확인 없이 공개키를 보내는 모두에게 같은 세션 키를 전달하는 서버를 사용한다. 만약 악의적인 목적의 앱이 해당 서버의 주소, 포트를 알고 접근하는 경우, 서버는 별다른 인증 없이 해당 소켓으로 세션 키를 전달 할 것이다. 이를 방지하기 위해선, PDU를 처리하거나, CA를 통한 인증 혹은 일종의 약속된 프로토콜을 통해 방지할 수 있을 것이다. 또한 PDU 없이 처리되는 과정에서 보내고자 하는 데이터의 양을 모르는 채 발신, 수신하기 때문에 데이터를 다 수신 했다는 보장이 없다. PDU를 정의하고 해당 PDU에 길이 정보를 넣는다면, 이를 보장할 수 있을 것이다. 또한 PDU에 암호 알고리즘을 상호 교환하거나, 카운터, TimeStamp 등을 정의한다면 Replay Attack이나 Dos 공격 등에 대응할 수도 있다고 생각한다. 이번 hSolution의 경우, 서버의 의존성이 강해 서버의 안정성이 요구되는데, 최소한 서버의 가용성을 공격하는 수단을 철저하게 방어해야 할 것이다.

서버의 경우, 회사 내부망 고정IP 환경에서 구동하였고, 서버의 IP와 포트의 경우 로컬에서 테스트하기 때문에 라이브러리에 하드코딩하였다. 추후에 이번 hSolution을 개선 및 확장해야 한다면 서버 또한 따로 구축해야하고 애플리케이션 별 서명 지문값을 저장할 데이터베이스 또한 구현이 필요할 것이다.

JVM과 리눅스, 윈도우 모두 리틀엔디안 환경이다. 네트워크로 내보내는 데이터는 빅엔디안이 산업 표준(네트워크 표준 바이트 순서)이라고 알고 있다. 서버의 OS나, 클라이언트의 OS 모두 엔디안이 일치하는 고정된 환경에서 구현하기 때문에 엔디안을 고려하지 않은 채 개발하였다. 만약 서버가 Solaris와 같은 빅엔디안 환경이라면 분명 이 부분을 고려하고 개발해야할 것이다. htonl(), ntohl()... 등과 같이 엔디안을 바꿔주는 함수를 사용하거나, unsigned char로 캐스팅하며 & 0xFF로 비트단위 연산을 하는 등

여러 방법이 있을 수 있다. Java와 호환이 좋고 네트워크에서 널리 쓰이는 솔라리스가 서버로 운영될 수 있기 때문에 분명 고려해야할 것이다.

2-5. 한계점

서버를 통한 애플리케이션 무결성 검증의 경우 한계점이 명확하다. hSolution을 거친 애플리케이션을 디컴파일링하여 주입한 hSolution 클래스를 제거하거나, hSolution 내의 검증 함수의 반환 값만 무효화 한다면 무결성 검증 없이 애플리케이션이 구동 될 것 이다. 이를 방지하기 위해 난독화 과정이 추가될 수 있지만, 난독화의 경우, 사람이 읽기 어렵게 만들 뿐 직접적인 디컴파일 방지 기법이 아니라고 생각한다. 악의적인 공격자가 충분한 시간만 투자하거나, 난독화 해제 툴 등을 통해 얼마든지 복구할 수 있다. 때문에 디컴파일을 해도 보이지 않는 Native Library를 적극 활용하거나, 서버를 통한 인증이 아니라면 애플리케이션의 보안은 힘들 것이다. 충분한 지식이 아니더라도, 디컴파일 툴이나, Smali Injection의 대한 많은 자료들이 오픈 소스로 존재하기 때문에 조금만 관심이 있더라도 충분히 앱을 위변조 할 수 있을 것이다.

이번 프로젝트의 타겟 애플리케이션의 경우 무결성에 대한 검증 과정이 없고, 오프라인으로 돌아가는 애플리케이션이기 때문에 이러한 Smali 코드 주입이 가능했고, 변조가 가능했다. 디컴파일 방지 기법을 찾아봤을 때 가장 흥미가 있던 자료는 디컴파일 시 생기는 classes.dex 파일 자체를 암호화 하는 방법이 있었다. 정상적인 경우 설치할 때 서버를 통해 서 해당 파일을 복호화하고, 서버의 인증을 받지 못한 경우 앱 자체의 classes.dex 파일이 암호화 되어있기 때문에, 해당 앱을 획득한 공격자가 디컴파일을 통해 앱을 뜯어보려 해도 위변조가 불가능 할 것이다. 또는 난독화 과정을 통해 classes.dex 파일을 \$1, \$2, \$1\$1, 등으로 분리하여 멀티텍스트로 처리하여 디컴파일 툴을 통해 보지 못하게 하거나, 함수와 변수 이름을 화면에 출력하지 못하는 문자로 바꾸어 처리하는 등 여러 기법들이 있었다. 이러한 모든 디컴파일 방지 기법의 경우, 서버를 타는 과정이 필수로 처리된다. 로컬에서만 구동되는 경우 충분히 조작이 가능하기 때문이라고 생각한다.

또한 이번 프로젝트의 경우 해당 타겟이 되는 앱에 해당하는 Smali코드, AndroidManifest.xml 파일만 작성되어있고, 자동화 되는 과정은 해당 코드와 파일들을 디컴파일된 애플리케이션 폴더에 넣는 것이다.

이런 식으로 작동 흐름을 예측하여 만드는 개발은 좋지 않은 개발임을 알고 있다. 다만, 이번 프로젝트에서 주입을 통해 이런 식으로 무결성 검증이 가능함을 보이기 위해 개발하였다.

미리 작성된 코드를 주입하는 것이 아닌, 기존 AndroidManifest.xml 을 토대로 앱 구성과 진입점을 파악 한 뒤, 상황에 맞는 Smali 코드를 작성되게 하고 해당 파일과 코드를 주입하는 것이 올바르다고 생각한다. 이러한 과정이 자동화되기 위해선 xml 파일의 파서도 필요하고 해당 xml 파일을 토대로 애플리케이션의 최초 진입점 파악도 중요하다. 애플리케이션 무결성 검증이 최우선시 되어야하기 때문이다.

이번 프로젝트의 경우 기존 안드로이드 애플리케이션이 임의로 구축한 리눅스 서버에 접속이 가능하고, 데이터 수신 발신을 통해 정상적인 값이 전달되며 이를 통해 무결성 검증이 가능한지 보는 테스트였기 때문에, 해당 타겟 애플리케이션에 해당하는 코드를 미리 작성해두었다.

만약 해당 프로젝트가 확장되어 기타 애플리케이션도 지원되게 하려면 해당 애플리케이션의 AndroidManifest.xml을 분석하여 최초 진입점을 찾고 해당 진입점에 실행 될 Smali 코드의 생성을 자동화 하는 과정이 추가로 필요할 것이다.

Java 코드는 디컴파일링이 쉬운 편이다. 거의 가능한 모든 부분이 디컴파일 가능하고, 난독화의 경우 함수명을 바꾼다거나 여기저기 찢어놓는다 해도 바이트 코드의 문자열(String)은 문자열 그대로 존재해야하는 등의 한계점이 있다. 그렇다고 모든 기능을 네이티브의 수준으로 내려 라이브러리 형태로 제공한다 해도, 라이브러리가 노출된다면 IDA hex-ray 등과 같은 툴로 디컴파일이 또 가능하다. 이렇듯 어느 단계라도 완벽한 보안은 존재하지 않는다. 어떤 조치를 취하더라도 시간이 지난다면 공격할 수 있는 방법이 분명 나올 것이다. 따라서 이러한 관점으로 봤을 때, 조금이라도 더 공격을 지체시키거나 꼬아놓는 등 시간을 지연시키거나 보다 더 많은 공격자의 노력이 필요하게 설계하는 것이 더 옳다고 생각하게 됐다. 널리 쓰이고 있는 RSA와 같은 경우도 소수의 경우의 수가 많아서 사실상 불가능이라는 의미가 결국 brute force가 가능하긴 하다는 말이지 않은가. 양자컴퓨터의 발전으로 소인수분해를 기반으로 뒀던 RSA도 쇼어 알고리즘을 통해 공격이 가능했고, 이번 현장실습 중 만들었던 PBKDF2의 경우도 인증용 테스트 벡터상의 해시 반복 횟수가 10만 번이었던 것처럼 완벽한 방어보단, 조금 더 귀찮게, 조금 더 지연되게 구상하는 것이 맞는 것 같다.

2-6. 오류코드

오류 출처	오류	오류 코드 번호	오류 코드 설명	비고
hSolution Library	HSOLUTION_ERR_VERIFY_INTEGRITY	1000	hSolution 검증 과정	
	HSOLUTION_ERR_SOCK_ERR	1001 (1000 + 1)	소켓 생성 오류	
	HSOLUTION_ERR_CONNECT_ERR	1002 (1000 + 2)	소켓 연결 오류	

	HSOLUTION_ERR_SEND_ERR	1003 (1000 + 3)	소켓을 통한 발신 오류	
	HSOLUTION_ERR_RECV_ERR	1004 (1000 + 4)	소켓을 통한 수신 오류	
	HSOLUTION_ERR_VERIFY_ERR	1005 (1005 + 5)	무결성 검증 실패	
EdgeCrypto Library	HSOLUTION_ERR_ASYM	2000	EdgeCrypto 비대칭 관련	관련 세부 사항은 EdgeCrypto 내부 처리
	HSOLUTION_ERR_ASYM_GEN_KEY_PAIR	2001 (2000 + 1)	비대칭 키 생성 오류	
	HSOLUTION_ERR_ASYM_DEC	2002 (2000 + 2)	비대칭 키 복호화 오류	
	HSOLUTION_ERR_SYMM	3000	EdgeCrypto 대칭키 관련	
	HSOLUTION_ERR_SYMM_ENC	3001 (3000 + 1)	EdgeCrypto 암호화 오류	
	HSOLUTION_ERR_SYMM_DEC	3002 (3000 + 2)	EdgeCrypto 복호화 오류	

3. 참고문헌

3-1. 참고문헌

- ◆ 애플리케이션 서명, 2022-09-13, Android Source
(<https://source.android.com/docs/security/features/apksigning?hl=ko>)
- ◆ SMALI SUMMARY, 2023-04-21, 정재의 Notion
(<https://seongjaeh.notion.site/SMALI-SUMMARY-1840cba821354c91803ed8b52a50cf40>)
- ◆ Android) ‘killProcess’로 앱을 종료하였지만 ... , 2021-01-11, Glacier’s Daily Log
(<https://h-glacier.tistory.com/104>)
- ◆ 2048, 2022-09-28, tpcstd
(<https://github.com/tpcstd/2048>)
- ◆ 이찬희, 정윤식 and 조성제. (2013). 안드로이드 애플리케이션에 대한 역공학 방지 기법. 보안공학연구논문지, 10(1), 41-50.
- ◆ 공개특허 10-2014-0029562, 바른소프트기술 주식회사 / 라온시큐어, 대한민국특허청, 안드로이드 어플리케이션의 디컴파일 방지를 위한 암호화 방법,
- ◆ 악성 APK을 이용한 Dynamic Dex Loading 분석, 2022-05-31, IGLOO
(<https://www.igloo.co.kr/security-information/%EC%95%85%EC%84%B1-apk%EC%9D%84-%EC%9D%B4%EC%9A%A9%ED%95%9C-dynamic-dex-loading-%EB%B6%84%EC%84%9D/>)
- ◆ Big-endian, Little-endian 빅 엔디안, 리틀 엔디안, 정보통신기술용어해설
(http://www.ktword.co.kr/test/view/view.php?m_temp1=2353)