**네이티브 라이브러리**

개발자가 작성한 다음 특정 컴퓨터가 아키텍처용으로 컴파일한 코드, 대부분 C와 C++로 작성된 코드

기존의 C/C++로 개발한 코드가 안드로이드 플랫폼에서 동작하는 바이너리로 빌드

🡺 그래픽 라이브러리와 같이 수학적으로 집약적이거나 시간에 민감한 작업을 위한 것

DEX 바이트코드 🡪 Java로 디컴파일 가능

컴파일된 코드 🡪 어셈블리로 분석

**JNI(Java Native Interface)**

개발자가 네이티브 코드로 구현되는 Java 메소드를 선언할 수 있음

Android NDK 🡪 JNI를 기반으로 하는 Android 전용 도구

Android 기본 라이브러리 🡪 .so ELF 파일 형식의 공유 객체 라이브러리로 APK에 포함

애플리케이션이 라이브러리를 메모리에 로드하기 위한 API 호출

1. System.loadLibrary(“calc”)

2. System.load(“lib/armeabi/libclac.so”)

두 API 중 하나가 실행되면 JNI\_OnLoad 네이티브 라이브러리 함수도 실행

Java 선언 네이티브 메서드 호출 🡪 네이티브 라이브러리의 페어링된 네이티브 함수가 실행

ex) public native String doThingsInNativeLibrary(int var0);

**페어링 하는 법**

1. JNI 네이티브 메소드 이름 확인을 사용한 동적 연결

2. RegisterNativesAPI 호출을 이용한 정적 링크

**동적 연결**

동적 연결을 위한 이름 지정

con.android.interesting.Stuff에서의 연결

Java\_com\_android\_interesting\_Stuff\_doThingsInNativeLibrary

네이티브 라이브러리에 해당 이름의 함수가 없으면 정적 수행 필요

**정적 연결**

개발자가 사양에 따라 기본 함수의 이름을 지정하기를 원하지 않거나,

지정할 수 없는 경우 정적 연결을 사용

jint RegisterNatives(JNIEnv \*env, jclass clazz, const JNINativeMethod \*methods, jint nMethods);

typedef struct {

char \*name;

char \*signature;

void \*fnPtr;

} JNINativeMethod;

**리버스 엔지니어링 – 정적 연결**

Java 선언 네이티브 메서드가 호출될 때 네이티브 라이브러리의 서브 루틴을 결정하기 위해 RegisterNative로 전달되는 JNINative 메서드 구조를 찾음

🡺 JNINatvie 메서드 구조에는 Java에서 선언한 네이티브 메서드 이름의 문자열과 메서드 signature의 문자열이 필요하므로 네이티브 라이브러리에서 찾을 수 있음

**Method Signature**

JNINative method 구조에는 method signature이 필요

🡺 인수 유형과 반환하는 인수 유형

Z 🡪 Boolean / B 🡪 Byte / C 🡪 Char / S 🡪 Short / I 🡪 Int / J 🡪 Long / F 🡪 Float

D 🡪 Double / L fully-qualified-class 🡪 fully-qualified-class / [ type 🡪 type[]

(arg-types) ret-type 🡪 method type / V 🡪 Void

실습 5 🡪 네이티브 메서드가 호출될 때 실행되는 네이티브 함수 식별

**과제 시행 순서**

1. jadx에서 Mediacode.apk 열기

2. jadx에서 선언된 모든 네이티브 메소드를 검색

3. 선언된 Native Method 주위에 Native Library가 로드된 곳이 있는지 확인, 이는 어떤 Native Library가 구현될지 알려줌

4. 새 디렉토리 생성 및 해당 디렉토리에 APK 복사하여 기본 라이브러리 추출, 다음 미디어코드 압축을 풀기

5. 분석하고자 하는 Native Library의 Architecture 선택

6. ghidraRun 명령어를 통해 ghidra 실행

7. “New Project”에서 “Non-Shared Project” 선택, 프로젝트를 저장할 경로 선택 후 이름 생성. 이는 Binary File로 load할 수 있는 프로젝트 생성

8. 생성하였다면, Code Browser를 열기 위해 dragon icon 선택, Native Library를 tool로 로드하기 위해 “File” 🡪 “Import File” 선택

9. “Analyze” 선택

10. linking 정보를 사용하여 Java-declared Native Mehtod가 호출되었을 때 실행될 Native Library의 함수 확인

**JNIEnv** : JNI 함수에 대한 함수 포인터의 구조, Android 기본 라이브러리의 모든 JNI 기능은 JNIEnv\*를 첫 번째 인수로 사용

🡺 JNIEnv와 JavaVM의 C 선언은 C++ 선언과는 다름. 둘 중 어떤 것에 포함되어 있느냐에 따라 “jni.h”는 각각 다른 typedefs를 제공한다. 따라서 두 언어에 포함되는 header file에는 JNIEnv 인자를 넣지 않는 것을 추천한다.

**android native library를 분석할 때, JNIEnv의 존재가 의미하는 것**

1. JNI Native Function의 경우, 인자들은 2만큼 이동할 것이다. 첫 번째 인자는 항상 JNIEnv\*이다. 두 번째 인자는 함수를 실행해야 하는 객체이다. static native mehtod의 경우에는 이것은 null이다.

2. 코드가 offset을 JNIEnv\*에 추가하고 해당 위치에서 함수 포인터를 가져오기 위해 역참조를한 다음 함수로 분기하기 때문에 disassembly에서의 간접적인 brach들을 보게될 것이다.

함수 포인터들이 각기 다른 offset들에서 무엇인지 알아보기 위한 JNIEnv 구조체의 C에 의한 구현 spreadsheet

🡺 수동 작업 외에, 모든 과정을 생략하고도 JNI 함수를 얻을 수 있음. IDA Pro 디컴파일러에서 JNI 함수의 첫 인자를 JNIEnv\* 형태로 다시 입력하면 호출되는 JNI 함수를 자동으로 식별함. Ghidra에서는 JNI types를 먼저 로드해야 함. VM의 ~/jni\_all.gdt에서 이용 가능함

1. Ghidra에서 사용할 수 있도록 불러오려면, “파일 보관 열기” 클릭

2. jni\_all.gdt 선택

3. ghidra에 로드되면 디컴파일러에서 인자 유형을 선택하고 “Retype Variable” 선택, JNIEnv\*로 선택. 이후에 표시됨

**앱의 동작 및 implementation을 숨기기 위한 난독화**

**🡺** 난독화를 풀기 전, 이 정보가 가치 있는 것인지를 판단할 것

**DEX ByteCode의 경우**

정적 난독화 해제 방법을 응용 프로그램에서 확인 후, 이를 java 파일로 복사하여 난독 처리된 부분에서 실행

**Java 및 Native Code**

Python 및 기타 스크립트 언어로 난독 해제 알고리즘 변환.

**난독화 표시자**

1. No-Strings : Java와 Android는 문자열 의존도가 높음. 따라서 난독화 가능성

🡺 문자열을 인수로 하는 method 호출을 찾고, 인수의 출처 파악. 문자열 인수는 string 인수를 사용하는 api로 전달되기 때문에 deobuscation 함수로 이동

2. Scrambled String : java 및 android api는 일반 텍스트 문자열을 필요로 함

🡺 스크램블 문자열은 api에 전달되기 전 모두 동일한 method로 전달됨

3. assets/디렉토리 및 dexclassloader 호출의 이진 파일 : 추가 코드 압축 풀기 및 로드 가능(dexclassloader를 통해 원격 다운로드도 가능)

🡺 파일을 읽어오는 위치 확인 후 경로 따라 올라감.

4. Native library JNI 함수 : native method를 실행하려면 jni가 native library의 함수를 java의 native method 선언과 쌍으로 구성할 수 있어야 함. 따라서 둘 중 하나가 어느 시점에 존재

🡺 JNI\_OnLoad method에서 시작하여 추가 코드를 로드하는 난독 제거 루틴 탐색