Binary lifting을 이용한 안드로이드 라이브러리 취약점 분석*

이성원, 윤종희
영남대학교 컴퓨터공학과
noke15@ynu.ac.kr, youn@yu.ac.kr

Android library vulnerability analysis using binary lifting

Sung-Won Lee, Jonghee Youn
Computer Engineering, Yeungnam University

요 약

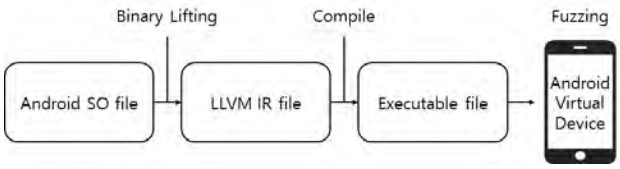
안드로이드 OS 는 대중적이고 중요한 시스템으로 자리 잡았고, 이에 따른 다양한 연구도 진행 중이다. 본 논문에서는 보안측면에서의 취약점 분석 방법을 제시하여, 각종 보안 위협을 예방하는데 기여하고자 한다. 안드로이드 라이브러리를 대상으로 Binary Lifting 기술을 사용하여 코드기반(LLVM IR) 퍼징을 진행하는, 취약점 분석 과정을 설계 수행한다.

1. 서론

스마트 폰과 IoT제품들 중 안드로이드 시스템은 높은 비중을 차지하고 있으며, 용도에 맞는 다양한 어플리케이션이 등장하고 개발되고 있다. 사용자의 편의를 제공할 뿐만 아니라 핵심적인 역할을 담당하는 만큼 보안사고가 발생했을 경우 개인이나 집단에게 매우 치명적일 수 있으며, 이를 방지하기 위한 노력과 연구가 반드시 필요하다. 본 논문에서는 안드로이드 시스템 중 라이브러리를 대상으로 Binary Lifting(본 논문에서는 바이너리를 LLVM IR[1]로 변환하는 과정을 의미한다) 하여 취약점을 발견하고 분석하는 방법을 제안한다.

2. 분석 프로세스 설계

안드로이드 라이브러리를 효율적으로 분석하기 위해 기존의 오픈소스 도구들을 결합하는 방식으로 진행되었다. MCSEMA[2], LLVM, Libfuzzer[3], Android Studio[4] 도구들을 중점적으로 사용하며, 취약점을 찾는 방식으로는 퍼징을 사용한다. 코드가 없는 바이너리 상태인 라이브러리를 LLVM IR 이라는 코드레벨로 변환하고, 이를 통해 소스코드를 사용하는 Libfuzzer로 퍼징 할 수 있게 된다. 그림 1은 전체적인 취약점 분석과정을 나타낸다.



<그림 1> 전체 취약점 분석 과정

각 도구마다 다양한 버전이 존재하고, LLVM과의 의존성 문제도 존재하기에 버전에 따라 각 도구들이 호환이 안 되는 상황이 발생하기도 한다. 본 논문에서 사용하는 환경과 도구의 버전과 표 1 과 같다.

<표 1> 분석에 사용된 환경과 도구의 버전

대상	환경 및 버전
Host	Ubuntu 18.04 x86_64
Guest	AVD - Android Pie x86_64
MCSEMA	LLVM 8
NDK	r21 Ver. x86_64 linux android clang

분석 프로세스는 Lifting, Compile, Fuzzing 3단계로 이루어진다.

Lifting 단계에서, 지원하는 도구들은 다양하게 개발되고 있으며, 그 중 안드로이드 시스템을 대상으로 하기에 x86_64 와 aarch64 아키텍쳐를 지원하

*이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2018R1D1A1B07050647)

는 MCSEMA를 사용하였다. MCSEMA에서 Lifting 하는 과정은 바이너리를 대상으로 Control Flow를 뽑고 Lifting 하는 2단계를 거쳐 bc 확장자를 가진 LLVM IR 을 생성한다.

Compile 단계에서, 생성된 LLVM IR 을 다시 실행 파일로 만들기 위해선 Clang이 필요하고, Host PC의 리눅스 환경에서 안드로이드 환경의 실행파일을 생성하기 위해 크로스 컴파일방법을 사용한다. 크로스 컴파일을 지원하는 Android Studio의 Ndk 도구를 사용하여 Guest 환경인 안드로이드 Pie x86_64 에서 실행 가능한 바이너리를 생성한다. r21 Ndk 버전에서 Clang 버전은 9.0.8로 설정되어 있으며, Libfuzzer를 사용하기 위한 옵션으로 -fsanitize=fuzzer,address를 인자로 적용하여 컴파일 한다. 안드로이드 시스템에 대한 또다른 컴파일 방법으로는 AOSP[5]를 사용하여 빌드 후 크로스 컴파일 하는 방법이 있다. 해당 방법의 경우 가장 높은 Clang의 버전이 6 버전으로(AOSP Android Pie 기준) 사용 하는 다른 도구와 LLVM 버전이 다르고, Libfuzzer 관련 라이브러리 또한 지금은 사용하지 않는 Libfuzzer.a를 사용한다.

Fuzzing 단계에서, 생성한 바이너리를 실행시킬 환경으로는 가상머신을 사용한다. Android Studio의 Android Virtual Device(AVD)를 사용하여 안드로이드 Pie x86_64 가상환경을 구축한다. 각 단계를 거쳐 생성된 바이너리를 실행시켜 퍼징을 통한 라이브러리 취약점 분석이 가능하다.

3. 실험 및 결과

실제 스토어에서 다운 가능한 어플리케이션을 대상으로 설계한 취약점 분석 프로세스를 진행하였다.

대상 어플리케이션의 APK파일 내부에서 분석하고자 하는 SO를 추출. 안드로이드 기기에서 작동하는 어플리케이션이라 추출된 SO 파일의 아키텍쳐는 aarch64로 확인된다. Lifting 후 Compile 과정 중에서 컴파일 되어 나오는 파일은 x86_64시스템으로 변경된다. MCSEMA에서 핵심 과정에 사용하는 라이브러리인 libmcsema_rt가 원인으로 x86, x86_64만 지원하고 있기 때문에 크로스 컴파일 과정 중 x86_64 아키텍쳐로 컴파일 한다. 다른 아키텍쳐로 크로스 컴파일 시 컴파일러가 incompatible target 에러를 출력한다.

완성된 실행파일을 AVD 환경에서 실행시키면 Libfuzzer를 사용하여 퍼징, 취약점을 분석한다. 그림 2와 3은 AVD환경에서 실행 했을 때 출력되는 결과 분석의 일부이다.

<그림 3> 취약점 분석 결과 1

<그림 2> 취약점 분석 결과 2

실행결과 Heap buffer overflow를 발생시키는 부분을 확인할 수 있다. 실제 어플리케이션에서 해당 라이브러리의 함수를 호출할 때 공격에 대한 방어가 존재할 수도 있지만, 라이브러리의 특성상 이후에도 계속 사용될 가능성이 높기 때문에 라이브러리 자체의 보안 취약점을 개선하는 것이 안전하다.

4. 결론

본 논문에서는 안드로이드 시스템의 라이브러리를 대상으로 하며 Binary Lifting을 통해 Libfuzzer로 퍼징하는 방법을 제안했다. 이러한 방식은 안드로이드 기본라이브러리를 대상으로 진행할 수 있을 뿐만 아니라 소스코드가 주어지지 않은 APK를 대상으로도 코드 기반(LLVM IR) 테스트를 통해 취약점 분석이 가능하다. 안드로이드 어플리케이션이 급증하고 다양한 기능을 위해 자체 제작한 라이브러리들이 포함되기 때문에 본 논문에서 제안하는 방식으로 취약점을 분석하고 개선하여 보안 위협을 줄일 수 있다.

참고문헌

[1] https://llvm.org/docs/LangRef.html
[2] https://github.com/lifting-bits/mcsema
[3] https://llvm.org/docs/LibFuzzer.html
[4] https://developer.android.com/studio
[5] https://source.android.com/