IoT 디바이스 펌웨어 취약점 탐지 방법의 비교 분석

김동준O 한선우 신기성 송미희 조진성
경희대학교 컴퓨터공학과

Comparative Analysis of IoT Device Firmware Vulnerability Detection Method

Dongjun KimO Seonwoo Han Kisung Shin Mihee Song Jinsung Cho
Department of Computer Science and Engineering, KyungHee University

요 약

IoT 디바이스 펌웨어를 모방하여 취약점을 보유한 샘플 코드를 작성하고 정적 분석, 퍼징, 기호실행 방법을 이용한 무료 도구들을 사용하여 취약점 분석을 실시한다. 결과의 비교 분석을 통해 각 탐지 방법론에 대한 특성을 조사하고, 분석 도구들의 장단점 비교를 통해 취약점 탐지 방안을 제시해본다.

1. 서 론

최근 사물 인터넷은 사회의 전 분야에 걸쳐 확장되고 있다. 사물 인터넷의 특성 상 수많은 IoT 기기들이 해킹의 대상이 될 수 있기 때문에 보안의 중요성이 강조된다. 특히, IoT 디바이스들은 임베디드 시스템으로서 오픈소스 펌웨어들을 기반으로 발전하고 있기 때문에, 기반 오픈소스들에 대한 취약점 분석을 통해 취약점들을 조기에 탐지하고 개선하는 연구가 필요하다.

우리는 동일한 코드를 대상으로 도구 간 효율적인 비교 분석을 할 수 있도록 IoT 서비스를 모방한 취약 코드를 직접 작성하였다. 그리고 다양한 취약점 분석 방법에 따라, 공개되어있는 여러 가지 취약점 탐지 도구들을 이용하여 각 도구와 방법들의 장단점을 비교분석하여 취약점 탐지 방안을 제시하였다.

본 논문의 구성은 다음과 같다. 2절에서는 다양한 소프트웨어 취약점 탐지 방법을 소개하고, 3절에서는 선정한 취약점 탐지 도구와 작성한 취약 코드에 대해 설명한다. 4절에서는 탐지 도구별로 작성된 코드에 대해 수행한 결과와 분석을 기술하며, 5절에서는 결론을 맺는다.

2. 취약점 탐지 방법 소개

2.1. 정적 분석

정적 분석이란 실제 실행 없이 컴퓨터 소프트웨어를 분석하는 것을 말한다. 일반적으로 초보적인 실수나 잘못된 코드 등 개발자가 고려하지 못한 상황으로 인해 취약점이 발생되며, 개발자의 document등의 구조분석으로는 탐지가 어렵기 때문에 정적 분석을 활용한다.

소스코드 취약점 분석 방법은 취약점 사례에서 수동으로 rule을 추출해내어 패턴을 비교하는 방식이기 때문에 엄청난 양의 개발된 코드들에 대한 커버리지의 부족으로 다른 방법론에 비해 높은 오탐율을 보였다. 그러나 최근 코드 복제 검증 방법(CLORIFI) 등을 이용하여 많은 보완이 이루어지고 있다.[1]

소스코드 자체를 기반으로 하는 정적 분석은 사전에 해당 코드 구조에 대한 이해에 대한 요구도가 낮으며, 동적 취약점 분석 등에 비해 분석도구의 사용이 간편하다는 장점을 가진다.
반면에 rule의 개수와 취약 패턴의 종류에 의존적이어서 단일 방법론으로서는 퍼징과 기호실행의 탐지범위를 커버하기 어렵다는 점이 대표적인 단점으로 꼽힌다.

2.2. 퍼징

퍼징은 무작위로 생성한 데이터를 입력하여 버퍼오버플로우(BOF), 포맷스트링버그(FSB), Use-After-Free 등과 같은 프로그램의 결함을 발견하기 위한 기법이다.

퍼징을 효율적으로 수행하기 위해서는 입출력 데이터 또는 대상 애플리케이션이 불러오는 파일 구조를 정확히 알고 있어야 하거나, 자동화된 도구를 이용해야 한다.

일반적으로 알려진 퍼징의 장점으로 입력파일 생성 규칙만 명시해준다면, 테스팅을 자동으로 진행하는 자동화와 입력값을 무작위로 대입하기 때문에 정확하진 않지만 모든 부분에 대해서 테스팅을 수행할 수 있는 높은 코드 커버리지를 들 수 있다. 단점으로는 무작위로

이 논문은 교육부 및 한국연구재단의
기초연구사업(NRF-2017R1D1A1B04035914)과
과학기술정보통신부 및 정보통신기획평가원의
SW중심대학 사업(2017-0-00093)의 지원으로 수행된
연구결과임.

입력데이터를 설정하기 때문에, 소프트웨어의 원하는 기능이나 함수의 취약점을 정확히 찾아내기 어렵다는 점과, 무작위로 입력데이터를 설정하기 때문에, 공격 가능한 취약점 식별을 보장할 수 없는 불확실성 및 무작위성이 있다.[2]

2.3. 기호 실행

기호 실행은 변수들을 이용해 악성 코드들의 분기를 찾아내는 기법이다. 기호 실행은 바이너리의 분기를 제약조건으로 보고 제약조건들을 풀어내는 과정을 거친다. 즉, 특정 소프트웨어 프로그램의 분기를 수식화하여 생성되는 분기점을 수행되도록 입력을 진행하는 것이다. 프로그램의 실행 단위를 분기로 구분하여 특정 조건에서 각 분기에 도달하는 과정을 수행시켜 정상 동작여부를 판별한다. 입력값에 대한 실행 경로를 가지고 있다면, 역으로 실행 경로에 접근하기 위한 입력값의 생성도 가능하기 때문에 취약점 탐색을 위한 위치까지 범위를 탐색하는 용도로 많이 사용된다.[3]

또한, 기존의 다른 동적 실행 기법에서는 특정 에러를 유발시키는 특정한 테스트 케이스를 직접 만들어내야 하지만 기호 실행을 활용하면 포인터 역참조와 같이 잠재적으로 위험한 명령에 대해 자동적으로 검출할 수 있게 된다.[4]

기호화를 위해서는 프로그램의 구조 파악이 선행되어야 하며, 분기가 수없이 나누어지는 프로그램은 기호 실행의 대상으로 적절하지 않다. 특정 프로토콜을 사용하는 프로그램의 경우 프로토콜의 잘못된 설계로 인해 발생하는 취약점은 탐지가 불가능하다. 해당 프로그램은 개발자가 의도한 방향으로 정상 동작하기 때문이다.

3. 취약점 탐지 도구 및 취약 코드 작성

3.1. 정적 분석

임베디드 환경에서 작동하는 IoT SW들의 특성상 c 또는 c++로 작성되었기 때문에 분석 도구 역시 c와 c++언어에 대하여 탐지할 수 있는 도구들로 선정하였다. 그리고 쉽게 접근하여 사용할 수 있는 오픈소스 도구들로 한정하였다. 이러한 기준에 따라 YASCA, Cppcheck, Clang static analyzer로 총 3개의 도구를 선정하였다.

3.2. 퍼징

프로그램에 무작위 데이터를 입력하여 버그 및 취약점을 찾아주는 자동화된 툴중 하나인 AFL은 현재 Linux netlink, libvorbis, PHP 등과 같이 오늘날 프로그램에서 수많은 취약점을 발견하는데 도움이 되는 오픈 소스 fuzzer이므로 AFL를 퍼징 도구로 선정하였다.

AFL은 테스트 케이스의 코드 적용 범위를 효율적으로 늘리기 위해 유전자 알고리즘을 사용하며 성능 오버헤드가 적고 다양하고 효과적인 전략 방식을 사용하여 퍼징을 수행한다.[5]

LibFuzzer는 화이트박스 테스트를 이용한 퍼저로써 입력값을 변경하여 코드 커버리지를 넓히는 스마트 fuzzer의 일종이다. LLVM 컴파일러 인프라 프로젝트를 통해 퍼징을 지원하며 함수의 반복 호출로 크래시를 발생시키는 방식으로 작동하게 된다. 바로 아래에 설명할 KLEE와 같이 LLVM이 서서히 GCC를 대체하는 프로젝트가 되어가며 중요성이 커지고 있으므로 해당 퍼징 툴을 선정하였다.

3.3. 기호 실행

기존 기호 실행의 한계점인 경로 폭발을 개선하기 위한 시도로서 기호실행과 실제 수행을 결합하여 실행 경로를 분석하는 콘콜릭 실행이 있는데, 이에 대한 대표적인 연구로 KLEE가 있다. 최근 LLVM의 영향력이 커짐에 따라 LLVM 기반의 KLEE를 기호 실행 도구로 선정하였다. KLEE는 심볼릭 프로세스의 운영체제와 인터프리터 역할을 동시에 수행한다. 심볼릭 프로세스는 State라고 표현하는데, 실제 콘콜릭 실행 경로를 따라가며 분기 조건을 저장하는 프로세스이다. 각 State는 프로세스와 동일하게 스택, 힙, 프로그램 카운터(PC) 등 의 정보를 갖고 있고, 자신이 수행한 경로 조건 정보를 축적해 두었다가 경로 종료 시 테스트 케이스 도출에 활용 한다.[6]

3.4. 취약 코드 작성

모든 도구를 실제 오픈소스 펌웨어에 대하여 적용하기에는 코드의 양이 매우 방대하여 비교분석에 적합하지 않았다. 그래서 우리는 각 탐지 도구별로 조사한 일반적인 특징들에 근거하여 동일한 소스 코드를 대상으로 효율적인 분석을 할 수 있도록 취약 코드를 직접 작성하였다. 실제 IoT 환경에 주로 이용되는 C언어를 사용하여 작성하였으며, IoT 스피커를 통한 홈 뱅킹 서비스를 모방하여 작성하였다.

작성된 코드는 main()함수와 아이디, 패스워드, 계좌번호, 계좌비밀번호 순으로 구성된 userInfo.txt파일로 구성된다. main()함수는 switch문을 기점으로 Use-After-Free 취약점을 포함하는 case 1과 Out-Of-Bound 취약점을 포함하는 case 2로 나뉜다. case 1에서는 송금 기능을 구현한 것으로, 사용자의 입력을 저장할 메모리를 할당한 후 송금할 계좌를 입력받은 뒤 그 정보를 sendToServer()라는 함수로 넘기게 된다. sendToServer() 함수는 main()함수에서 선언한 메모리 할당을 해제하고, 동일한 크기로 서버에서 사용할 메모리를 할당한다. 이 부분에서 Use-After-Free 구조를 갖게 되는 코드이며, 이후 처리중인 함수가 맞는지 여부를 사용자에게 입력받고 정상적인 경우 처리 완료 메시지를 출력하는 함수가 호출된다. 그러나 사용자가 비정상적인 입력으로 관리자함수의 포인터를 입력하게 되면 관리자 함수가 실행되어 모든 유저 정보를 노출시키는 공격이 가능해지는 코드로 작성하였다.

case 2에서는 계정 삭제 기능으로 사용자에게 정말로 삭제할지 여부를 확인하는 단계적 과정을 함수로 구현하였다. 사용자의 입력이 제대로 이루어졌는지를 검사하는 if문과 그 내부에 다음 단계의 함수를 호출하는 구조로 중첩 if문의 구조를 갖는다. 최종 단계에서 사용자에게 탈퇴를 위한 서비스 만족도 질문에 선언된 배열의 인덱스를 초과하는 크기의 입력이 들어오게 되면 비정상적인 접근이 발생하여 Out-Of-Bound 오류로 프로그램이 종료되는 구조이다..

AFL과 libFuzzer 그리고 KLEE의 경우 분석을 수행하기 위해 printf나 scanf 등의 함수를 도구 전용 함수로 교체하는 등의 수정을 하였으나 구조적인 변동은 없이 탐지에 사용하였다.

4. 취약점 탐지 방법 비교 분석

본 논문에서 작성한 취약 코드에 대한 3가지 방법의 탐지 결과를 표로 나타낸 결과는 다음과 같다.

도구	방법	case 1 탐지	case 2 탐지
YASCA	정적 분석	O	X
Cppcheck	정적 분석	O	X
Clang static analyzer	정적 분석	O	△
libFuzzer	스마트 퍼징	O	X
AFL	스마트 퍼징	X	X
KLEE	기호 실행	X	O

[Table 1.] 분석 방법에 따른 분석 도구 비교 결과

YASCA, Cppcheck, Clang static analyzer를 이용한 정적 분석 결과 세 도구 모두에서 case 1의 Use-After-Free 오류에 대해서는 성공적으로 탐지하는 것을 확인할 수 있었다. 그러나 프로그램 자체가 중지될 수 있는 case 2의 Out-Of-Bound 오류에 대해서는 YASCA와 Cppcheck에서는 탐지하지 못하였고, Clang static analyzer에서는 Out-Of-Bound 부분에서 발생할 수 있는 Use-After-Free를 탐지하였다. 또한, 작성된 취약 코드에서 의도한 오류는 2가지였으나 YASCA에서는 총 13가지, Cppcheck에서는 스타일 지적을 포함하고 파일입출력부분에서 Out-Of-Bounds 오류가 발생한다는 3가지 오탐을 포함한 총 16가지, Clang static analyzer에서는 총 12가지를 보고하는 등 실제 오류에 비해 보고 사항이 너무 방대하고 오탐 또한 존재하며, 3가지 도구에서 보고에 사용하는 용어가 각기 달라 취약점 탐지에 혼란을 야기할 수도 있음을 확인하였다.

AFL를 이용한 분석 결과 case 1, case 2의 오류에 대해서는 올바른 input을 넣어주는 testcase를 생성하였을 때만 탐지가 가능했으나 비슷하지 않은 testcase를 생성하여 퍼저를 돌렸을 때 AFL 퍼저의 무분별한 testcase 변환으로 조건문을 뚫고 들어갈 수 없음을 알 수 있었다. 이를 통해 input 값에 대해 길이가 길고 복잡한 경우 데이터를 바이트 단위로 분할하여 퍼징을 하는 libfuzzer를 사용하는 것이 좀 더 효율적일 수 있음을 확인하였다.

libFuzzer는 case 1의 경우에 대해 퍼징 데이터가 생성되면, 그 데이터를 바이트단위로 분할하여 아이디, 비밀번호, 계좌번호를 전부 뚫고 Use-After-Free 취약점을 발견 해 낼 수 있었다. case 2의 경우 각 조건문의 일치여부에 대해 퍼징 데이터와 비교를 했는데, 조건문이 다수 중첩으로 적용되어 있는 경우 그 내부까지 뚫고 들어갈 수 없음을 확인할 수 있었고, 이를 통해 코드 커버리지가 낮다는 것을 확인할 수 있었다. 코드 커버리지 문제를 해결하기 위해 기호실행이 같이 수행되어야 할 것이다.

KLEE의 경우 case 1을 탐지하기 위해서는 파일에서 유저 정보를 한줄씩 읽어와 아이디와 패스워드를 대조해야 한다. 하나의 파일을 읽더라도 그 안의 모든 유저 정보와 비교할 때 수많은 분기가 생기게 되어 path explosion 문제를 피할 수 없다. 반면 case 2의 경우 중첩된 if 문을 만족하는 정확한 값을 모두 알아야 문제가 있는 코드에 접근이 가능하므로 퍼저에 비해 해당 부분에 쉽게 접근할 수 있다. 퍼저가 접근하지 못하는 부분을 접근할 수 있도록 정확한 값을 알아내어 그 내부를 퍼징하는 것에 도움을 주는 방향으로 활용할 수 있다.

5. 결론 및 향후 연구

IoT의 빠른 발전을 방해하는 요소로 보안 문제가 대두되고 있다. 우리는 그러한 보안 문제들을 조기에 파악하고 조치하는 데에 일조하고자 IoT 서비스를 모방한 취약 코드를 작성하고 다양한 취약점 분석 도구들의 장단점을 분석하였다. 분석 결과, 정적 분석의 경우 쉽게 접근할 수 있으나 오탐과 너무 많은 보고 사항으로 인해 단독으로 사용하기에는 비효율적이었다. AFL의 경우 직접적인 코드 수정이 필요하지 않아 libFuzzer에 비해 빠르고 간편하게 분석을 시작할 수 있으나, 사용자에게 입력으로 testcase를 적절하게 줄 수 있는 직관이 없을 경우 제대로된 탐지가 불가능한 성능을 보였다. libFuzzer의 경우 코드를 상당부분 직접적으로 수정해야 하지만 어느 정도의 중첩 if문을 통과하여 오류 검출을 할 수 있었다. KLEE의 경우 굉장히 많은 중첩 if문의 경우에도 취약 코드가 있는 조건문에 대한 경로를 발굴할 수 있었으나, while등의 반복문에 취약한 path explosion 특징을 보였다. 따라서, 실제 오픈소스 등 거대 코드에 대하여 취약점 탐지를 수행할 경우 정적 분석은 퍼징이나 기호실행을 할 범위를 한정하는 데에 참고하고, AFL 등의 퍼징을 사용하는 것이 효과적이다. 코드의 입력 구조가 복잡한 경우 AFL보다는 libFuzzer를 이용하는 것이 더 좋은 탐지 성능을 보일 수 있으며, 중첩 if문 등의 희박한 확률로 발생하는 취약점에 대응하기 위한 방법으로는 KLEE가 효과적으로 사용될 수 있다.

향후 연구사항으로는 Double-Free 오류 등 다양한 취약점에 대한 도구간의 비교 분석, 오픈소스 이외에 상용 탐지 도구들에 대한 비교분석을 통해 더 효율적인 도구적 협업을 구상할 수 있을 것이다.

6. 참고 문헌

[1] 이홍제, “Software vulnerability discovery using code clone verification”, 고려대학교 박사학위논문, 2017

[2] 임기영, 강성훈, 김승주, "퍼징 기반의 상용 및 공개 소프트웨어에 대한 보안약점 진단 방법 연구." 정보보호학회지 제26권 제1호, 2016

[3] 오상환, “보안 취약점 자동 탐색 및 대응기술 동향,” 정보보호학회지 제28권 제2호, 2018

[4] C.Cadar, “Symbolic execution for testing complex software,” 스탠포드 대학교 박사학위논문, 2009

[5] “american fuzzy lop(2.52b)”, 2019.04.28, URL: http://lcamtuf.coredump.cx/afl/

[6] 오상환, “SW 보안 취약점 자동 탐색 및 대응 기술 분석,” 한국산학기술학회논문지 제18권 제11호, 2017