악성코드 Argument Detection 방법 연구*

황신운, 윤종희
영남대학교 컴퓨터공학과
hswoon@ynu.ac.kr, youn@yu.ac.kr

A Study of Malware Argument Detection

Shin-Woon Hwang, Jonghee Youn
Dept. of Computer Science, Yeungnam University

요 약

악성코드 분석방법의 발전에 따라 악성코드의 분석우회기법도 나날이 발전하여 대량의 악성코드분석이 다양한 이유로 수행되지 않고 있다. 대부분의 악성코드는 소스코드가 없는 바이너리로 동적 분석이 동작하지 않는 원인을 파악하기 어렵다. 동적 분석이 실행되지 않는 악성코드들은 입력 값에 따라 악성코드가 동작하거나, 특정 시간대를 일치하는 등 다양한 트리거가 존재한다. 본 논문에서는 트리거가 필요한 악성코드에 대해 바이너리 리프팅(lifting) 기술을 활용한 새로운 동적 분석방법을 제안한다. 바이너리 리프팅 기술은 소스코드가 없는 바이너리를 LLVM IR 로 변환시키는 기술로서 이를 활용해 입력 값 유무에 따른 악성코드를 판별하고자 한다. 전달인자를 사용하는 코드와 사용하지 않는 코드간 LLVM IR 을 비교분석하여 전달인자에 따른 악성코드 동작 여부를 판별해 대량의 악성코드 동적 분석시스템의 분석률을 높이는 방안을 제안하고자 한다.

1. 서론

악성코드는 사용자의 의사와 이익에 반해 시스템을 파괴하거나 정보를 유출하는 등 악의적 활동을 수행하는 프로그램이다. 악성코드 분석시스템은 크게 정적 분석과 동적 분석으로 이루어져 있다. 악성코드의 분석 방법이 발전됨에 따라 악성코드 또한 분석을 피하는 방법도 나날이 발전하여, 악성코드가 의도적으로 악성행위를 실행하지 않아 악성코드 분석 시스템의 동적 분석이 진행되지 않는 경우들이 있다. 대표적인 케이스로 DDL(Data Definition Language) 파일이 존재하지 않는 경우, 필요로 하는 특정 파일이 존재하지 않는 경우, 사용 환경이 다른 경우, 인자가 필요한 경우, 가상머신을 탐지하는 경우 등 다양한 조건들로 악성 행위를 분석하기 힘들게 한다. 본 논문에서는 이러한 경우를 트리거라고 분류하고 대표적인 트리거 중 하나인 인자가 필요한 경우에 대한 바이너리 파일을 McSema 을 이용하여 변환한 LLVM IR 에서의 전달인자 탐지법을 제시하고자 한다.

2. 배경 지식

2-1 분석 시스템에서 동적 분석이 되지 않는 이유

동적 분석은 분석 시스템이 구축한 가상환경에서 동작한다. 하지만 악성코드를 분석하는 중에 특정 기준을 만족하지 못해 동작하지 않는 경우가 있다. 예를 들어 정해진 날짜와 시간 후에만 악성코드를 실행하여 해당 시간 이전에는 탐지를 못하도록 한다. Trojan.Agent.544256 악성코드의 경우 'sandbox', 'malware' 등의 파일명이 있을 경우 자가 종료한다. 또한 현재 실행되고 있는 프로세스들을 검색하여 안티 바이러스 및 분석 툴의 실행 여부를 확인하고 가상환경을 탐지한다.[1] Trojan.Agent.Ripper 악성코드의 경우 커맨드라인에 전달하는 인자에 따라 각각 다른 동작을 수행한다. 첫번째 인자는 주요 동작을 결정하고 두 번째 인자는 하위 동작을 결정한다.[2] 이처럼 동적 분석이 어려운 악성코드들이 존재하며 본 논문에서는 이를 트리거라고 분류한다.

2-2 바이너리 변환 기술

LLVM 은 현재 가장 활발하게 개발되고 있는 오픈소스 컴파일러 중 하나다. LLVM 은 frontend, optimizer, backend 로 구성되어 있다. Frontend 에서 고급 언어(HLL)를 읽어 파싱한 후 LLVM IR(Intermediate representation 중간표현)으로 바꿔주며[3] 이를 이용한

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

연구가 수행되고 있다. 그중에서도 완성된 바이너리 파일을 특정 아키텍처에 맞게 수행할 수 있도록 하기 위한 연구가 수행되고 있는데, 이를 바이너리 리프팅 기술이라고 한다. 대표적으로 McSema 가 있으며, 바이너리 파일을 LLVM IR 로 변환시켜 원하는 아키텍처에 맞게 재컴파일을 가능하게 한다.[4] 본 논문에서는 이러한 바이너리 변환 기술을 이용하여 트리거 탐지 법을 제시하고자 한다.

3. 본론

전달인자가 필요한 경우의 코드의 특징 중 하나는 argc 값을 특정 값에 비교한 후 악성행위를 실행하도록 하는 것이다. 본 연구에서는 McSema 로 바이너리 파일을 LLVM IR 로 변환한 코드에서 argc 를 탐지하는 것을 테스트 목표로 두었다. 테스트 방법은 3 단계로 나뉜다. 첫 번째는 argc 값을 이용하는 테스트 코드와 사용하지 않는 테스트 코드를 각각 바이너리 파일로 컴파일한다. 다음은 각각의 바이너리 파일은 McSema 를 이용하여 LLVM IR 로 리프팅 시켜준다. 마지막은 바이너리 형식으로 저장된 LLVM IR 파일을 가독성이 좋은 텍스트 파일로 Clang 을 이용하여 컴파일하여 해당 파일에서 argc 사용 여부를 탐지한다.

분석 방법의 적절성을 검증하고자 본 연구에서 제안하는 방법을 따라 테스트 코드를 직접 작성하여 바이너리 파일로 변환한 후 테스트를 진행했다. 그림 1 예제 코드는 argc 사용 여부를 확인하기 위한 간단한 테스트 코드로 LLVM IR 파일에서 분석을 용이하게 하기 위해 비교 값을 흔하지 않은 수 '7072' 를 사용했다.

```
define internal %struct.Memory* @sub_1140_main(%struct.State* noalias nonnull %state,
inst_1149:
  %0 = load i32, i32* @RDI_2296_224d390
  %1 = load i64, i64* @RBP_2328_224d3a8
  %2 = load i64, i64* @RSP_2312_224d3a8, align 8, !tbaa !1240
  %3 = add i64 %2, -8
  %4 = inttoptr i64 %3 to i64*
  store i64 %1, i64* %4
  store i64 %3, i64* @RBP_2328_224d3a8, align 8, !tbaa !1216
  %5 = sub i64 %3, 16
  store i64 %5, i64* @RSP_2312_224d3a8, align 8, !tbaa !1216
  %6 = sub i64 %3, 4
  %7 = inttoptr i64 %6 to i32*
  store i32 %0, i32* %7
  %8 = load i64, i64* @RSI_2280_224d3a8
  %9 = inttoptr i64 %5 to i64*
  store i64 %8, i64* %9
  %10 = load i32, i32* %7
  %11 = sub i32 %10, 7072
  %12 = icmp ult i32 %10, 7072
```

(그림 1) LLVM IR 파일 1

그림 1 은 argc 를 사용하는 코드를 제안한 방법에 따라 테스트한 결과다. LLVM IR 을 분석한 결과 일반 소스코드를 LLVM IR 로 변환한 결과와 달리 문자열 'argc'를 직접적으로 찾아볼 수 없었다. LLVM IR 명령어 중 비교 명령어인 'icmp'와 정수 '7072'를 통해 인자 argc 사용을 예측할 수 있었다. 그림 1 에서 RSP 레지스터의 주소값을 연산하고 해당 주소에 포함된 데이터 값을 정수값 7072 와 비교하는 모습을 확인할 수 있었다. 다른 예제코드를 테스트한 결과인 그림 2 와 그림 1 은 주소값 연산 operand 값의 차이를 제외하고 매우 유사함을 알 수 있었다. 인자 argc 를 사용하지 않는 예제 코드를 대상으로 테스트하여 분석한 결과 입력받는 함수를 사용할 경우 RAX 레지스터값을 바로 사용하는 차이가 존재했다. 이러한 결과로 LLVM IR 비교 명령어 'icmp'의 첫 번째 operand 값을 역으로 추적해서 argc 값의 위치를 확인할 수 있을 것이다.

```
define internal %struct.Memory* @sub_11c9_main(%struct.State* noalias nonnull %state,
inst_11c9:
  %0 = load i32, i32* @RDI_2296_1818390
  %1 = load i64, i64* @RBP_2328_18183a8
  %2 = load i64, i64* @RSP_2312_18183a8, align 8, !tbaa !1240
  %3 = add i64 %2, -8
  %4 = inttoptr i64 %3 to i64*
  store i64 %1, i64* %4
  store i64 %3, i64* @RBP_2328_18183a8, align 8, !tbaa !1216
  %5 = sub i64 %3, 32
  store i64 %5, i64* @RSP_2312_18183a8, align 8, !tbaa !1216
  %6 = sub i64 %3, 20
  %7 = inttoptr i64 %6 to i32*
  store i32 %0, i32* %7
  %8 = load i64, i64* @RSI_2280_18183a8
  %9 = inttoptr i64 %5 to i64*
  store i64 %8, i64* %9
  %10 = sub i64 %3, 16
  %11 = inttoptr i64 %10 to i64*
  store i64 0, i64* %11
  %12 = sub i64 %3, 8
  %13 = inttoptr i64 %12 to i64*
  store i64 0, i64* %13
  %14 = load i32, i32* %7
  %15 = sub i32 %14, 7072
  %16 = icmp ult i32 %14, 7072
```

(그림 2) LLVM IR 파일 2

4. 결론

본 논문에서는 악성코드 중 특정 기준을 만족하지 못하여 동적 분석시스템 내에서 동작하지 않는 경우를 트리거로 지칭했다. 트리거의 종류는 명령어를 입력받거나, 임의 시간에 동작 및 IP 와 통신 등 다양한 경우가 있었으며 그중 인자가 필요한 경우에 리프팅 기술을 활용해 탐지하는 방법을 제안했다. 논문 속 실험을 통해 argc 값이 리프팅된 LLVM IR 값에서 위치를 찾을 수 있었다. 이러한 방법은 바이너리가 필요로 하는 입력 값의 카운트를 기반으로 임의의 값을 대입해 동적 분석을 한 단계 더 진행시킬 수 있을 것으로 판단되며, 자동 분석되지 않는 대량의 악성코드를 분석할 수 있도록 하는 개선방안이 될 수 있다. 이후 연구에서는 입력 값의 카운트뿐만 아니라 argv 값을 비교하는 LLVM IR 코드의 특징군을 찾아내 동적 분석의 정확도를 높여보고자 한다.

참고문헌

[1] Trojan.Agent.544256 악성코드 분석 보고서

[Website].(2018). URL: https://blog.alyac.co.kr/1993

[2] 악성코드 분석리포트 Trojan.Agent.Ripper

[Website].(2016). URL: https://blog.alyac.co.kr/868

[3] LLVM http://www.aosabook.org/en/llvm.html

[4] McSema https://github.com/lifting-bits/mcsema