Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.33, NO.2, Apr. 2023 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2023.33.2.253

효율적 데이터 의존성 분석을 이용한 바이너리 기반 Null Pointer Dereference 취약점 탐지 도구*

김 문 회,[1†] 오 희 국[2‡]
[1,2]한양대학교 컴퓨터공학과 (대학원생, 교수)

Efficient Null Pointer Dereference Vulnerability Detection by Data Dependency Analysis on Binary*

Wenhui Jin,[1†] Heekuck Oh[2‡]
[12]Department of Computer Science and Engineering, Hanyang University
(Graduate student, Processor)

요 약

널 포인터 역참조 (Null Pointer Dereference) 취약점은 정상적인 메모리 주소를 저장한 포인터가 아닌 널 포인터를 참조할 때 발생하는 취약점이다. 그러므로, 서비스거부공격 (Denial-of-service)와 같은 공격에 악용되어 큰 피해를 줄 수 있기 때문에 식별하고 제거해야 할 중요한 취약점이다. 기존 연구에서는 기호실행과 같은 정적분석을 통해 취약점을 탐지하는 방법을 많이 제안하였다. 그러나 커널과 같은 복잡도가 높은 대형 프로그램을 검사할 때는 경로폭발 (path explosion)과 제약조건(constraint solver) 때문에 효율성이 제한되며 주어진 시간 내에 탐지하지 못할 수 있다. 또는 대형 프로그램 중 일부 함수들 혹은 라이브러리 함수를 탐지할 때 전체 프로그램의 상태를 알 수 없기 때문에 완전한 분석을 수행하지 못해 정확도가 높지 않은 경우가 있다. 본 논문에서는 소스코드가 없는 대규모 프로그램에서 특정한 일부 기능 검사 할 때 빠르고 정확하게 검사하기 위한 가벼운 취약점 탐지도구를 연구개발 하였다. 변수나 포인터를 기호로 표시하고 프로그램 실행을 에뮬레이션하여, 각 실행경로에서 데이터 의존성 (data dependency) 분석과 휴리스틱 방법으로 널 포인터 역참조 취약점을 탐지한다. 기존 bap_toolkit과 실험하여 비교하였을 때 오탐율이 8% 높지만, 존재하는 취약점을 모두 탐지해냈다.

ABSTRACT

The Null Pointer Dereference vulnerability is a significant vulnerability that can cause severe attacks such as denial-of-service. Previous research has proposed methods for detecting vulnerabilities, but large and complex programs pose a challenge to their efficiency. In this paper, we present a lightweight tool for detecting specific functions in large binary programs through symbolizing variables and emulating program execution. The tool detects vulnerabilities through data dependency analysis and heuristics in each execution path. While our tool had an 8% higher false positive rate than the bap_toolkit, it detected all existing vulnerabilities in our dataset.

Keywords: NUll Pointer Dereference, Vulnerability Detection, Static Analysis, Symbolic Execution, Binary Analysis

Received(10. 21. 2022), Modified(1st: 12. 22. 2022, 2nd: 02. 15. 2023), Accepted(02. 27. 2023)
* 본 논문은 2023년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (NRF-2022R1A2C2007255).
† 주저자, jinwenhui@hanyang.ac.kr
‡ 교신저자, hkoh@hanyang.ac.kr(Corresponding author)

I. 서 론

널 포인터 역참조 취약점은 정상적인 메모리 주소를 저장하는 포인터가 아닌 널 포인터가 참조할 때 발생하는 메모리 오류와 관련된 취약점이다. 그러므로, 서비스거부공격과 같은 공격에 이용되어 큰 피해를 줄 수 있기 때문에 식별하고 제거해야 할 중요한 취약점이다. Table 1은 2021년 가장 위험한 25가지의 취약점 유형 평가 결과이며 해당 취약점은 15위에 위치해 있다. 2022년 12월까지 CVE 사이트에 보고된 상용프로그램이나 라이브러리에서 발견된 취약점은 176 건이며[1], 널 포인터 역참조 취약점은 해커들에게 이용되어 큰 피해를 발생시키기 때문에 본 논문에서 해당 취약점 유형을 다룬다. 그뿐만 아니라, 본 논문에서 제안한 널 포인터 역참조 취약점을 탐지하는 과정에서 사용하는 바이너리 분석 방법은 메모리 경계 넘어 값 일기/쓰기 (Out-of-bounds Write/Read), 해제된 메모리 접근 (Use-after-free) 등의 취약점 탐지에도 유용하며, 후속 연구를 수행하기에 의미가 있다.

Table 1. 2021 CWE Top 25 Most Dangerous Software Weaknesses[1]

Rank	ID	Name	Score
[1]	CWE-787	Out-of-bounds Write	65.93
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84
[3]	CWE-125	Out-of-bounds Read	24.9
[4]	CWE-20	Improper Input Validation	20.47
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	19.54
[7]	CWE-416	Use After Free	16.83
[8]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.69
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	14.46
[10]	CWE-434	Unrestricted Upload of File with Dangerous Type	8.45
[11]	CWE-306	Missing Authentication for Critical Function	7.93
[12]	CWE-190	Integer Overflow or Wraparound	7.12
[13]	CWE-502	Deserialization of Untrusted Data	6.71
[14]	CWE-287	Improper Authentication	6.58
[15]	CWE-476	NULL Pointer Dereference	6.54

일반적으로 소스코드에서 취약점이 존재하는지 탐지해야 하지만 소스코드뿐만 아니라 바이너리에서도 탐지할 필요가 있다. 예를 들어, 컴퓨터나 휴대폰은 소프트웨어 업데이트를 쉽게 진행할 수 있지만 인프라에서 사용한 소형 사물인터넷 (IoT; Internet of Things) 임베디드 단말기나 오래 사용된 구형 기기 같은 경우 일부 기간이 지나면 더 이상 패치 서비스를 제공하지 않은 경우가 많다. 또는 원데이 (1-day) 취약점이 발생했을 때 소스코드가 없는 상황에서 바이너리에서 취약점을 빠르게 탐지하는 방법이 필요하다.

소프트웨어는 컴파일러가 소스코드를 컴파일해서 생성된 바이너리다. 컴파일러 자체에 취약점이 존재한다면 소스코드에서 완벽히 검사하더라도 취약점이 발생할 수 있다. 실제로 컴파일러 때문에 바이너리에서 백도어 취약점이 발견된 사례가 있다[2]. 그러므로, 바이너리에서 취약점을 탐지하는 작업은 소프트웨어 안전에 필수적인 방법이다. 본 논문에서 탐지할 대상은 바이너리 코드이며 다양한 아키텍처 기반의 바이너리에서 하나의 방법으로 취약점을 탐지하기 위해 바이너리를 먼저 VEX[3] 중간언어로 변환한 후 중간언어 코드를 기반으로 정적분석하여 취약점을 탐지한다.

기존 바이너리를 대상으로 취약점을 탐지하는 방법은 동적 방법, 정적방법과 동적 혼합 방법으로 나뉘어 있다[4-8]. 퍼징 (fuzzing) 은 대표적인 동적 탐지 방법이며 실제 프로그램을 실행하여 탐지한다. 여러 입력으로 실행 과정에서 오류가 발생했을 때 해당 오류를 유발하는 취약 코드의 존재 여부를 분석한다. 따라서 분석된 결과를 신뢰할 수 있으며 실제 실행을 통해 발견한 오류이기 때문에 탐지 결과의 정확성에 대해 증명할 필요도 없다. 그러나 퍼징과 같은 동적 실행 방법의 한계점은 프로그램 클수록 입력할 수 있는 범위가 너무 넓어 설정한 시간제한을 넘어도 프로그램에 있는 모든 가능한 실행경로를 탐색하지 못하는 것이다.

기호실행과 같은 정적 분석 탐지 방법은 프로그램 코드의 문법과 문맥을 분석하여 각 실행경로와 관련된 입력을 계산하여 높은 프로그램 커버리지를 가진 탐지를 수행할 수 있다[9-18]. 정적 분석의 한계점은 프로그램을 분석할 때 필요한 프로그램 실행 정보가 없기 때문에 포인터와 메모리 분석의 안전성과 완전성을 동시에 만족하지 못하여 방법에 따라 오탐

(FP: False Positive)과 미탐 (FN: False Negative)이 발생할 수 있다. 특히 프로그램 크고 복잡한 경우에는 경로 폭발이나 제약조건 해결 문제로 인해 깊은 경로까지 탐색하지 못 할 수 있다.

정적 탐지 방법과 동적 탐지 방법은 탐지의 수요에 따라 선택하여 사용하거나 결합하여 사용된다[19-20]. 본 논문에서 해결하고자 하는 문제는 복잡하고 규모가 큰 프로그램이나 라이브러리에서 전체 내용을 분석하여 취약점을 탐지하는 것이 아니라, 특정 기능과 관련된 한정된 범위에서 단일 함수에서 프로시저 내 분석 (intra-procedural) 또는 함수 간 프로시저 간 분석 (inter-procedural) 분석을 통해 널 포인터 역참조 취약점을 찾는 것이다.

예를 들어, CVE-2017-5668[21] 취약점은 libpurple라는 라이브러리에서 발견된 널 포인터 역참조 취약점이다. Libpurple[22]는 Pidgin에서 개발한 온라인 채팅 기능을 지원하는 라이브러리이며 Discord, Slack, WhatsApp 등 많은 도구에서 libpurple 기반 통신 서비스를 지원 및 사용한다. 해당 취약점은 함수 외부에서 파라미터로 포인터 'data'를 받아 참조하는데 Null인지를 체크하지 않고 사용하여 취약한 코드이다. 이러한 취약한 코드는 매년 지속해서 발견되고 있으며, 2022년 12월까지 176건이 CVE 사이트에 보고되었다.

라이브러리 코드나 대규모 프로그램에서 특정한 기능만을 검사할 때 외부에서 받은 포인터가 메모리에서 어떻게 처리되는지 분석하는 것이 가장 어려울 것이다. 이러한 포인터는 스택 변수, 일반적인 힙 공간, 복잡한 구조체로 되어 있는 힙 공간 등 다양한 형태로 되어 있기 때문이다. 그뿐만 아니라, 소스코드로 분석했을 때 포인터의 유형으로 구조체의 구성과 크기를 볼 수 있지만, 바이너리에서는 해당 정보가 없어서 분석하기에 한계가 있다. 기호실행 기반 방법은 완전한 프로그램을 분석할 때 많이 사용되지만, 본 논문에서 다루는 문제의 범위는 전체 프로그램이 아니기 때문에 외부에서 불러온 포인터의 대한 메모리 정보를 알 수 없기 때문에 정확하게 분석하기 힘들다. Bap[23] 기반 에뮬레이터를 이용한 도구를 실험해본 결과에 따르면 이런 경우 해당 도구는 임의 입력으로 임시 실행해보고 취약하지 않다고 판단한다. 임의의 입력으로 취약한 경로 찾는 방법은 대 규모 프로그램에서는 제안 시간 내 수행할 수가 없다.

```
static gboolean prplcb_xfer_new_send_cb(gpointer data, gint fd,
b_input_condition cond)
{
        PurpleXfer *xfer = data;
        struct im_connection *ic = purple_ic_by_pa(xfer->account);
        struct prpl_xfer_data *px = xfer->ui_data;
        PurpleBuddy *buddy;
        const char *who;
        buddy = purple_find_buddy(xfer->account, xfer->who);
        who = buddy ? purple_buddy_get_name(buddy) : xfer->who;
        px->ft = imcb_file_send_start(ic, (char *) who, xfer->filename,
        xfer->size);
+       if (!px->ft) {
+               return FALSE;
+       }
        px->ft->data = px;
```

Fig. 1. Vulnerable code (white) and Patch code (green) in CVE-2017-5668

본 논문은 많은 정보가 제한된 상태에서 취약점을 탐지하기 때문에 외부에서 들어온 값 또는 외부에서 수정할 수 있는 값 모두 널 포인터로 가정하여, 함수 간 제어 흐름과 데이터 흐름을 분석한다. 분석한 후 외부 포인터와 해당 포인터로 영향을 받는 포인터들 모두 추적대상으로 지정하여 역참조하는 코드 존재한지 확인한다.

그러나 해당 방법만으로 탐지하면 외부 마라미터이지만 널 아닌 파라미터 역참조하는 코드도 모두 취약하다고 판단되며 오탐이 많이 발생한다. 오탐을 줄이기 위해 해당 역참조 코드가 존재한 실행경로에서 if 문으로 해당 포인터가 널 포인터 인지 판별하는 코드 존재한지를 추적하고 해당 널 포인터가 아닌 경로에 존재한 역참조 코드는 취약하다고 판단하지 않으므로 오탐을 줄인다. 본 논문의 핵심 아이디어는 위와 같으며, 바이너리 상에서 적정 분석이 정확하게 이루어져야 취약점이 정확하게 탐지된다. 바이너리 정적분석하는 과정에서 탐지 정확성에 영형을 주는 요소 및 해결하는 휴리스틱 방법은 4장에서 서술되어 있다.

본 논문에서 기여한 내용은 다음과 같다.

1) 중간언어 VEX의 해석 도구 연구 및 개발: 제어 흐름과 데이터 흐름을 분석하기 위해 VEX 언어 코드를 문법적으로 해석하여 실제로 어떤 명령과 매칭 되는지 해석하는 도구를 개발하였다.

2) 실행 경로에 민감한 데이터 흐름 및 의존성 분석기능 연구 및 개발: 바이너리나 중간언어 코드에서 프로그램의 기호 실행이 어려운 부분은 메모리값과 포인터 값을 정확하게 분석해야 하는 문제이다. 소스코드처럼 변수명 정보가 없기 때문에 프로그램 실행 흐름에 따라 같은 레지스터가 어떤 메모리 영역을 가리키는지 분석해야 한다. 또는 변수 간의 값 전달뿐만 아니라 실제로 서로 다른 포인터이지만 동일한 메모리공간을 가리키는 포인터인지도 분석한다. 따라서

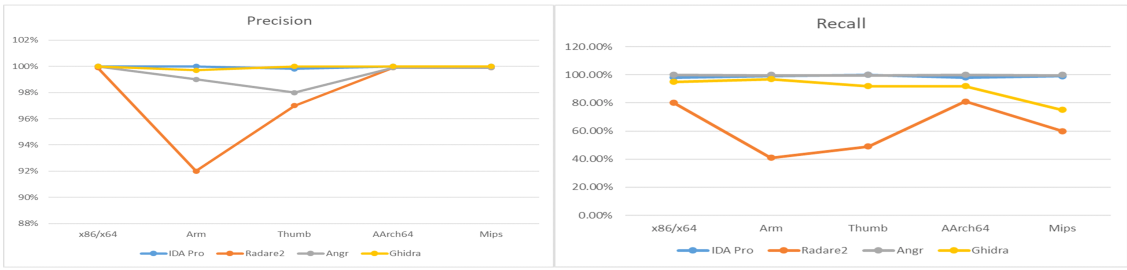

Fig. 2. Comparison of disassembly accuracy for different architectures among popular disassemblers

경로에 민감한 데이터 흐름 및 의존성 분석을 연구 및 개발하였다.

3) 프로시저 간 (Inter-procedural) 분석 연구 및 개발: 바이너리 코드는 소스코드와 달리 변수가 존재하지 않기 때문에 특정 주소와 값을 확인하는 방법은 레지스터를 확인하는 방법밖에 없다. 그러나 프로그램 실행 흐름에 따라 레지스터값이 계속 변경되어 이를 기록해야 하는 문제가 있다. 따라서 함수 호출의 경우 호출함수 (caller)와 피호출함수 (callee) 간 프로시저 변경, 상태 업데이트 및 복원 기능을 고려하여 레지스터 및 메모리값의 변화를 정확하게 추적하고 분석하는 기능을 연구하고 개발하였다.

4) 포인터 앨리어싱 (aliasing) 분석: 취약점 탐지의 정확성을 위해서는 포인터 앨리어싱 분석의 정확성이 매우 큰 영향을 준다. 그러나 기존 연구 Dtaint[19] 에서 제시한 포인터 앨리어싱 분석 방법은 명시적인 포인터만 찾아내고 코드 상에 존재하지 않은 암시적인 포인터 앨리어싱를 찾지 못하였다. 본 논문에서 이를 개선하여 함수 간 포인터 앨리어싱 분석 기능을 연구 및 개발하였다.

5) 널 포인터 역참조 취약점 탐지 휴리스틱: 기본적인 실행경로에 민감한 데이터 의존성 및 데이터 흐름 분석을 모두 수행한 후, 본 논문에서 제시한 문제를 해결하기 위한 널 포인터 역참조 취약점을 식별하기 위한 휴리스틱 방법연구 및 개발.

II. 배경 지식

예시 Fig.3.처럼 널 포인터 역참조 취약점을 탐지할 경우 우선 믿을 수 없는 데이터를 'source'로 지정하고 메모리 참조 등과 같은 위험한 명령어를 'sink'라고 지정한다. 이후 믿을 수 없는 데이터가 프로그램 실행경로를 따라 전파하는 과정을 추적한다. 실행 과정에서 'sink'에 접근하기 전에 아무런

```
0x401269:   endbr64
0x40126d:   push    rbp
0x40126e:   mov     rbp, rsp
0x401271:   sub     rsp, 0x10
0x401275:   mov     qword ptr [rbp - 8], 0      source
0x40127d:   cmp     qword ptr [rbp - 8], 0
0x401282:   setne   dl
0x401285:   mov     rax, qword ptr [rbp - 8]
0x401289:   mov     eax, dword ptr [rax]        sink
0x40128b:   cmp     eax, 5      Dereferencing Null !
0x40128e:   sete    al
0x401291:   and     eax, edx
0x401293:   test    al, al
0x401295:   je      0x4012a6
```

Fig. 3. Null pointer dereference vulnerability detection (assembly)

널 포인터인지 판별하는 제약조건 없는 경우 취약점이 있다고 판단한다. 예시에서 나온 코드는 가장 간단한 코드이며, 실제 프로그램의 경우는 매우 복잡하여 분석하기 어렵다.

2.1 VEX 중간 표현 언어

예시 Fig.4.에서 보인 것과 같이 Angr[3]에서는 VEX 중간 표현 언어를 사용하여 모든 아키텍처의 바이너리를 동일한 형식으로 정확하게 변환한다.

본 연구에서는 도구의 실용성을 고려하여 모든 아키텍처의 바이너리를 분석할 필요성이 있다. Fig.2.는 시중에 가장 인기가 많은 바이너리 분석 도구의 역어셈블리 정확성을 비교한 결과이다. 대부분의 도구에서 x86/x64 아키텍처의 경우 높은 정확성을 보이지만, 다른 바이너리 아키텍처를 분석할 경우 도구별로 많은 차이를 보였다. Bap은 본 논문실험 과정에서 간접 점프 (Indirect jump) 주소를 잘 계산해내지 못하는 문제를 발견하여 본 논문에서 제외하였고, IDA Pro[24]는 정확성이 높지만 상용도구이

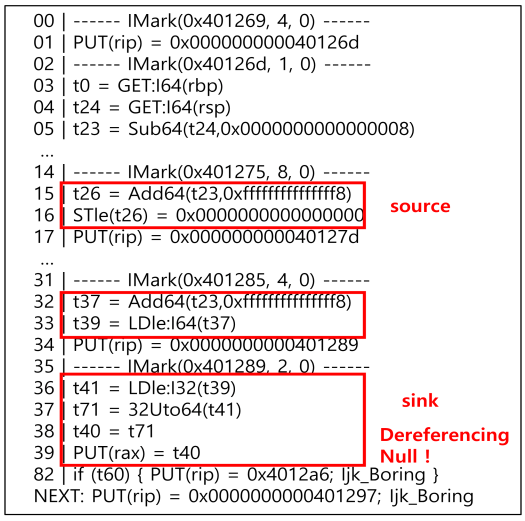

Fig. 4. Null pointer dereference vulnerability detection (VEX IR)

기 때문에 실용성 측면을 고려하여 선택하지 않았다.

본 논문에서는 Angr에서 제공하는 중간언어인 VEX가 범용성 측면에서 가장 적절한 것으로 판단하여 사용하게 되었다.

2.2 데이터 흐름 분석

정적분석에서 데이터의 흐름을 분석하기 위해서는 우선 제어흐름을 분석한 이후에 진행한다. 그러나 정적 분석의 경우 프로그램의 동적 실행정보를 알 수 없다. 예를 들어, 'if' 조건문에서 사용자 입력과 같은 외부 변수가 존재한 경우 어떠한 경로로 분석해야 하는지 판단하지 못한다. 경로에 민감하지 않은 분석 방법으로 실행하게 된다면 프로그램 분석을 진행할수록 각 변수의 범위는 무제한이 된다. 값의 변화는 널 포인터 역참조 취약점 분석의 정확성에 큰 영향을 미친다. 본 논문에서는 기호실행과 유사한 방식으로 알 수 없는 변수에 대해 기호로 표시하고 기호표현식을 바탕으로 에뮬레이션을 수행한다. 따라서 if 문에서 갈라지는 각 경로에서 기호의 표현식이 서로 다르다.

III. 관련 연구

기존의 널 포인터 역참조 취약점과 관련된 다양한 탐지 방법이 존재하며 기호실행은 대표적인 정적 탐지기법이고 퍼징은 대표적인 동적 탐지기법이다. 본 장에서는 널 포인터 역참조 취약점 유형을 탐지하기 위해 기존 연구에서 사용한 방법에 관해 서술하고 본 논문에서 제시한 방법과의 차이점을 비교한다.

3.1 퍼징 방법 (Fuzzing)

퍼징은 바이너리에서 프로그램의 버그나 취약점을 찾을 때 많이 사용하는 동적 탐지 방법이다. 퍼징 기술은 프로그램의 입력을 무작위로 생성하여 프로그램에 존재하는 모든 실행경로를 실행해보고 입력값에 따라 해당 실행경로에 오류가 나는지 확인하여 취약점을 탐지하는 방법이다. 퍼징 기법은 동적 탐지 기술이기 때문에 탐지의 정확성이 보장되며 탐지 결과에 대해 따로 증명할 필요는 없다.

퍼징 방법에서는 특정한 시간 내에 최대한 많은 실행경로를 분석실행하는 것이 목표이다. 가장 중요한 문제는 입력값을 생성하는 것이다. 프로그램에서 실행경로가 많이 존재하는 이유는 'if', 'while' 명령어와 같은 명령어이며 바이너리에서는 'JE'와 같은 조건식 명령어들이다. 조건식에 외부 입력이 들어가 있는 경우 입력을 무작위로 테스트하면 계속 같은 경로에 테스트하여 효율적이지 않은 경우가 존재한다. 예를 들어, "if (a 〈 10000)" 코드일 경우 a 값이 0에서 9999까지 입력하는 것은 프로그램의 새로운 경로를 탐색할 수 없다. 따라서 퍼징 기술이 발전함에 따라 변경기반 (mutation-based)의 퍼징과 생성기반 (generation-based) 방법을 활용하여 실험 데이터를 실행경로 탐색에 더욱 유리한 방향으로 변경하거나 주어진 템플릿 포맷에 따라 데이터를 생성해서 실험한다. 예를 들어, AFL[25]은 구글(Google)에서 개발한 대표적인 퍼징 도구다. AFL 기반으로 개선된 퍼징 도구가 상당히 많은 것으로 알려져 있으며, 퍼징의 정확성 및 가용성을 개선하는 연구가 계속 수행되고 있다.

그러나 본 논문에서 해결하고자 하는 문제는 라이브러리에서 존재하는 함수나 프로그램의 일부 함수들이기 때문에 퍼징 방법은 적절하지 않다. 또한 대형 프로그램을 테스트하면 주어진 시간 내에 취약한 코드가 존재하는 경로를 찾지 못할 가능성이 높다. Emulator[26]를 통해 프로그램을 분석하여 의심스러운 값을 널로 지정해서 테스트하는 방법도 존재하지만 본 연구에서는 빠르고 가벼운 방법을 제안하므로 효율성과 자동화 부분에서 차이가 있다.

3.2 기호 실행 (Symbolic Execution)

Saluki[27], Bap_toolkit, Angr 도구의 경우 모두 정적 기호실행 기반의 분석이 가능하다. 기호실행 방법은 정적 분석으로 외부 입력값을 모르기 때문에 기호로 표시되고 프로그래밍 코드에 따라 특정 코드지점 까지 프로그램 메모리, 변수 등 상태를 기호 표현식으로 표현한다. 그리고 특정 코드 지점까지 도달할 수 있는 입력 표현식으로 구성된 제약조건을 해당 조건식으로 풀어서 입력값을 구한다. 따라서 해당 입력으로 실행해보고 취약점이나 버그를 검증할 수 있어 많이 연구되고 사용되고 있다.

기호실행 방법이 이론적으로는 이점이 많지만 실제 복잡한 프로그램에 적용했을 때는 다음과 같은 한계점이 존재한다. 기호실행의 가장 핵심적인 아이디어는 변수를 기호로 표현하여 실행할 수 있는 경로를 모두 탐색하여 해당 경로를 실행할 수 있는 제약조건을 만족가능성 모듈 이론 해결기 (SMT solver; Satisfiability Modulo Theories solver)를 사용해 실제 입력값을 구하는 것이다. 그러나 프로그램이 너무 크고 복잡한 경우에는 반복문 loop, 재귀함수호출 recursive call, 등의 이유로 실행할 수 있는 경로가 기하급수적으로 커지는 경로폭발 (path explosion) 문제가 존재한다. 그러므로, 정해진 시간 내에 분석이 종료하지 못 한 경우가 많다. 두 번째로, 제약조건이 모두 선형이 아니기 때문에 비선형 조건이 많이 존재하는 실행경로의 폴지 못 하거나 지정한 시간 내 분석하지 못할 가능성이 있어 깊은 실행경로는 탐색하지 못하는 한계점도 존재한다.

기호실행은 메모리 상태를 포함한 프로그램 상태를 계속 유지하면서 실행을 에뮬레이션을 수행하는 강력한 분석 방법이다. 본 논문에서 해결하고자 하는 문제는 라이브러리에서 존재하는 함수나 대형 프로그램 중 일부 기능을 실현한 함수들이기 때문에 외부에서 파라미터로 받은 포인터에 대해 메모리 상태를 모르는 상태에서 분석하기 어렵다. 뿐만 아니라, 소스코드에서는 구조체 명을 통해 해당 포인터가 가리키는 메모리 공간의 크기와 내부 메서드 크기를 알 수 있지만 바이너리에서는 이와 관련된 정보가 없어 분석하기 더욱 어렵다.

3.3 유사한 기존 연구 비교

LUKE[28]는 소스코드 기반 널 포인터 역참조 취약점을 탐지하는 도구다. LUKE에서 제안하는 방법의 접근 방향은 본 논문에서 제한하는 방법의 접근 방향과 유사하다. 모두 오염분석 방식으로 데이터 의존관계를 분석한 후 의심스러운 지점에서부터 포인터를 역참조하는 코드까지 도달하는지 분석하여 취약점을 탐지하는 방법이다.

LUKE는 기호실행 방법이 아니지만 각 명령어를 기호로 표시하여 특정 프로그램 코드 지점까지 도달하기 위해 실행된 명령어를 수집한다. 수집하는 과정에서 조건판별식을 만난 경우에는 조건식의 값이 True, False인 경우 따로 구분하여 갈려진 경로를 다르게 표현하게 된다. 기호표현식으로 표현된 명령어 집합을 수집한 후 SMT 솔버로 해당 경로와 관련된 제약조건을 풀 수 있는지를 확인하여 도달할 수 없는 경로를 제외하여 FP를 줄인다.

요약을 하면, LUKE에서는 변수와 명령어의 관계를 모두 기호표현식으로 표시하여 의심스러운 실행경로가 존재하면 관련된 기호표현식들을 SMT 솔버로 실행이 가능한지를 증명하여 가능하면 취약한 실행경로가 있다고 판단한다.

본 논문에서는 제안하는 접근 방향은 LUKE와 비슷하며 모두 데이터 간의 의존관계를 분석하여 특정한 경로에서 의심스러운 포인터가 역참조되는지 탐지하는 것이다. 그러나 본 논문에서 해결하고자 하는 문제는 전체 프로그램 상태가 완전하지 않은 라이브러리 함수 혹은 완전한 프로그램 중 일부 함수들이다. LUKE 방법에서 오탐를 줄이는 방법은 포인터 분석 (points-to analysis)이 정확하다는 가정에서 제안한 방법이다. 포인터 앨리어싱은 프로그램 분석이나 취약점 탐지에 가장 크게 영향을 주는 중요한 요소다. 그러나 프로시저 실행 환경이 완전하지 않은 상태에서 일반적인 메모리 분석이나 포인터 분석이 어렵다. 특히 바이너리에서는 구조체 유형이 없기 때문에 해당 포인터로 인한 메모리 영역 접근과 관련된 포인터 에일리어싱은 정확하게 찾는 것은 더욱 어렵다. 본 논문에서는 스택과 일부 힙 범위에서 포인터 에일리어싱을 찾는 휴리스틱을 제안 및 개발하였다.

두 번째로, 본 논문에서 제안하는 방법은 먼저 실행이 가능한 경로를 따로 나눠 각 단일 실행경로별로 분석한다. 기호실행과 비슷하게 프로그램 내에서 정의한 변수는 실제 값으로 전파되고 외부 입력은 기호로 표시하므로 기호표현식으로 실행경로에 따라 프로그램 상태를 기록 및 업데이트한다. If, while과 같은 분기명령어를 만난 경우 관련된 변수는 해당 실행

```
void test ( gpointer p, int a){
        student *c = p-> student

        while (a < 10){
          c = NULL;
        }
        d = c->id
}
```

Fig. 5. Null pointer dereference vulnerability detection (assembly)

경로에 맞는 조건이 기호표현식으로 표시된다. 따라서 단순한 데이터 간 의존성 분석 방법보다 제어흐름과 결합한 분석방법은 암시적인 데이터 간 의존관계를 찾게 된다.

예를 들어, Fig.5.에서 변수 c의 명시적인 앨리어싱은 포인터 p이지만 변수 a의 값에 따라 c의 값은 Null일 수도 있다. 따라서 a와 c 사이에 직접적인 값을 할당하는 명령어는 없지만 제어흐름 상 간접적인 의존성이 있다. LUKE는 GVDG (Guarded Value Dependency Graph)에서 제안한 방법도 두 명령어 간 관계를 계산하므로 해당 경우를 찾을 수 있다. 그러나 LUKE에서는 기호의 값을 계산하지 않고 명령어 간의 표현식만 계산한다고 명시되어 있고 바이너리 분석하는 경우에는 적절하지 않는다. 본 논문에서 제안하는 방법은 바이너리 대상으로 포인터 에일리어싱을 정확히 분석하기 위해 계산할 수 있는 기호의 값은 모두 계산한다.

세 번째로, 바이너리에서 취약점을 탐지하기 위한 디스어셈블리의 정확성은 Angr에서 정확하게 추출했다고 가정을 하지만, VEX 코드의 특성에 맞도록 탐지 도구를 개발할 때 소스코드 분석과 다른 어려움이 존재하였다. 이에 자세한 내용은 4장에 있다.

IV. 제안 방법

논문의 연구 범위는 라이브러리와 대형 프로그램의 일부 기능을 실행하는 함수들에서 널 포인터 역참조 취약점을 탐지할 때 오탐을 낮추고 빠르게 탐지하는 것이다. 프로그램을 처음부터 분석하는 것이 아니기 때문에 외부에서 파라미터로 받은 포인터의 경우 관련된 메모리 상태에 대한 정보가 없기 때문에 기존의 기호실행과 같이 완전한 프로그램 경우 주어진 시간 내에 취약점을 탐지하는 것이 적절하지 않다. 본 논문에서 제안하는 방법의 핵심적인 아이디어는 메모리 모델링을 기호표현식으로 표시하고 프로그램 실행 경로에 따라 각 명령어를 실행했을 때 변수와 메모리 상태를 업데이트하면서 변수 간 의존관계를 분석한다. 단일 함수 또는 여러 함수 간 데이터 의존성을 분석한 후 의심스러운 포인터를 역참조하는 코드가 유효한지 검증하여 취약점을 탐지한다.

본 장의 내용은 1절 취약점 탐지 규칙, 2절 메모리 및 포인터 처리, 3절 데이터 의존성 분석과 4절 한계점 분석으로 구성되어 있다.

4.1 취약점 탐지 규칙

널 포인터 역참조에 취약하다고 판단하는 기준은 다음과 같다: 역참조 할 포인터가 Null 포인터이거나 Null 포인터일 수 있다면 취약하다고 판단한다. 단일 함수 내에서 프로시저 내 분석하는 경우와 함수 간 프로시저 간 분석 경우가 적용할 규칙이 일부는 다르며 구체적인 판단규칙은 아래와 같다.

- 믿을 수 없는 값 소스 (source): 1) scanf, read, recv 등 라이브러리 함수는 외부에서 입력받는다. 따라서 해당 함수의 파라미터로 쓰이는 포인터는 Null 값으로 받을 수 있다. 또한 함수 리턴 값을 받는 포인터도 Null 값을 받을 수 있어 소스로 본다. 2) 라이브러리 바이너리를 탐지하는 경우 export로 표시된 함수들은 호출자함수가 모르기 때문에 파라미터를 소스로 본다. 3) 탐지하는 함수 외에서 정의된 전역변수는 바이너리의 data 영역에서 값을 확인할 수 있지만 다른 함수가 실행하는 과정에서 해당 값을 수정할 수 있기 때문에 소스로 정한다. 단일 함수 분석이나 함수 간 분석에 모두 동일하게 위에서 서술한 규칙을 적용한다. 추가로, 단일 함수를 분석하는 경우 다른 함수를 분석할 수 없기 때문에 타 함수를 호출에 쓰이는 포인터와 실행 종료로 반환한 포인터도 소스로 본다.

- 위험 가능 제거 (sanitizer): 1) if 명령어에서 source와 관련된 변수가 Null인지 비교하는 명령어가 존재하는 실행경로에는 해당 포인터와 같은 메모리 주소를 가리키는 포인터들은 더 이상 소스로 추적하지 않는다. 기존 연구에서 오탐이 존재하는 이유 중 하나는 같은 주소를 가리키는 다른 포인터를 동일하게 처리하지 않아서 오탐이 발생한 것이다. 또한 Null 포인터인지 검사하는 조건식 명령어가 존재하지만 True인 경우 해당 포인터를 역참조하는 취약

코드도 존재하며 문맥을 분석하지 않고 단순히 문법적으로 분석하는 방법은 오탐이 발생한다. 2) 프로그램을 시작했을 때 소스로 표시되었지만 실행과정에서 유효한 메모리 주소값을 받는 경우가 있다. 이러한 경우 더 이상 소스로 보지 않는다.

- 취약점이 발생하는 지점 싱크 (sink): 널 포인터 취약점 경우 Null를 역참조하면 안 되기 때문에 deref(exp) 명령어를 싱크로 정의한다. 해당 명령어가 발견되면 exp가 소스인지 판단한다. 프로그램 실행경로에 따라 sanitization 검사를 모두 수행한 후 해당 명령어에 사용된 포인터가 소스로 표시되어 있다면 취약한 코드 있다고 판단한다.

4.2 메모리 및 포인터 처리

바이너리에서는 포인터라는 변수 유형이 존재하지 않는다. 따라서 메모리에 접근할 때 열어 명령어로 "(base 주소 + offset)" 방식으로 메모리에 접근한다. 멀티 포인터나 구조체 같은 경우에는 "deref(base 주소 + offset)* "을 여러 번 반복하여 메모리에 접근한다. 논문에서는 포인터 역참조를 "deref"로 표시한다.

라이브러리나 대형 프로그램 내 일부 함수를 분석할 때 파라미터로 외부에서 가져온 포인터에 대한 메모리 상태를 전혀 모르기 때문에 해당 메모리와 관련된 코드에 대해 정확한 상태 변화를 분석하기 어렵다. 예를 들어, Fig.6. 구조체의 경우 소스코드에서는 해당 구조체의 정의를 통해 할당된 메모리 공간이 204 바이트를 가진다는 것을 계산할 수 있다. 그러나 바이너리에서는 관련 정보가 없기 때문에 공간의 사이즈를 확인할 수 없다. 따라서 base + offset인 경우 해당 메모리의 유효한 범위와 구체적으로 어떤 메서드 영역에 가리킨 것인지 알 수 없다.

완전한 메모리 상태를 계산할 수 없기 때문에 본 논문에서는 완전한 메모리 상태를 분석하는 것이 목적이 아니라, 바이너리에서 포인터를 표현할 때 base 주소는 반드시 알아야 하는 특징을 이용하여 base 기호와 관련된 표현식으로 메모리 상태를 추적 및 업데이트한다. 예를 들면, Fig.6.에서 book 유형인 포인터를 파라미터로 받았을 때 rdi 레지스터로 전달받는다. 따라서 해당 메모리 공간 주소로 메모리 접근할 때는 deref (rdi + offset) 방식으로 접근하며 author에 있는 내용에 접근하려면 deref (deref (rdi + 50)+ offset) 식으로 표현한다.

```
struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} book;
```

Fig. 6. Structure example

실제 분석하는 과정에서는 구조체의 정보가 없어 사이즈 50을 알 수 없지만, 포인터 간 데이터 의존관계를 구할 수 있는 것은 장점이다. 동일한 힙 공간이나 stack 공간을 가리키는 포인터라면 base 주소 부분이 같기 때문이다.

4.3 데이터 흐름 분석

데이터 흐름을 분석할 경우 코드 실행에 따라 프로그램의 상태 (레지스터, 메모리 등) 값을 저장하면서 분석을 수행한다. 실행 경로에 민감한 분석 (path-sensitive)을 수행하기 때문에 if 문을 만날 때마다 두 가지 경로를 분석해야 하므로 n 개의 if 명령어가 있을 때 가장 나쁜 경우에는 전체 경로의 종류수가 까지 될 수 있어 분석에 아주 큰 오버헤드가 생길 수 있다.

이를 극복하기 위해서 본 논문에서는 프로그램 데이터흐름을 분석하여 먼저 모든 실행 가능한 경로를 수집한다. 모든 실행 경로 중 한 개라도 취약점이 발견되면 해당 프로그램은 취약하다고 판단한다.

그러나 정적분석의 경우 반복문 조건식을 계산할 경우 실행하지 않으면 알 수 없는 변수 값이 사용될 수 있다. 이러한 경우 코드를 몇 번 반복 실행해야 할지 정확하게 계산할 수 없다. 본 논문에서는 데이터 간 의존관계를 분석하기 때문에 Fig.7.와 같이 경로 순회 (path traversal) 방법으로 CFG 중 모든 엣지를 커버하기 위해 loop를 최소 두 번까지

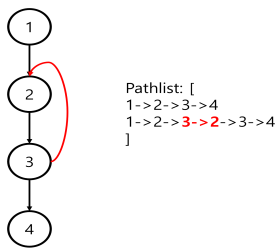

Fig. 7. Path exploration in loop

{'func_off_blkid': [('goodB2G', '0x5a', '0x401452')], 't5': ['bp'], 't4': ['t5-0xc'], 't2': ['deref(t4)'], 'cc_op': ['0x7'], 't15': ['t2'], 't6': ['t15'], 'cc_dep1': ['t6'], 'cc_dep2': ['0x0'], 't18': ['t6'], 't19': ['0x0'], 't17': ['t18<=t19'], 't16': ['t17'], 't13': ['t16'], 't20': ['t13'], 't8': ['t20'], 'Jmp_Cond_from_0x401456': [('0x401428', 't8', {}), ('0x401458', '!t8', {})]}	('func_off_blkid', [('goodB2G', '0x5a', '0x401452')]) ('t5', ['bp']) ('t4', ['t5-0xc', **'bp-0xc**']) ('t2', [**'deref(bp-0xc)',** 'deref(t4)']) ('cc_op', ['0x7']) ('t15', [**'deref(bp-0xc)',** 't2', 'deref(t4)']) ('t6', ['t15', **'deref(bp-0xc)',** 't2', 'deref(t4)']) ('cc_dep1', ['t6', **'deref(bp-0xc)',** 'deref(t4)', 't15', 't2']) ('cc_dep2', ['0x0']) ('t18', ['t6', **'deref(bp-0xc)',** 'deref(t4)', 't15', 't2']) ('t19', [**'0x0**']) ('t17', ['t2<=**0x0**', **'deref(bp-0xc)**<=**0x0**', 't18<=t19', 't15<=0x0', 't6<=0x0', 'deref(t4)<=0x0']) ('t16', ['t2<=0x0', **''deref(bp-0xc)**<=**0x0**', 't17', 't18<=t19', 't15<=0x0', 't6<=0x0', 'deref(t4)<=0x0']) ('t13', ['t2<=0x0', **'deref(bp-0xc)**<=**0x0**', 't17', 't18<=t19', 't15<=0x0', 't16', 't6<=0x0', 'deref(t4)<=0x0']) ('t20', ['t2<=0x0', **'deref(bp-0xc)**<=**0x0**', 't17', 't18<=t19', 't15<=0x0', 't13', 't16', 't6<=0x0', 'deref(t4)<=0x0']) ('t8', ['t2<=0x0', **'deref(bp-0xc)**<=**0x0**', 't17', 't18<=t19', 't15<=0x0', 't13', 'deref(t4)<=0x0', 't16', 't6<=0x0', 't20']) ('Jmp_Cond_from_0x401456', [('0x401428', **'deref(bp-0xc)<=0x0**', {}), ('0x401458', **'!deref(bp-0xc)<=0x0**', {})])

Fig. 8. VEX analysis result (left) and symbolic value propagation example (right)

경로에 추가한다.

분기문이 없는 실행경로를 모두 생성한 후 Fig.8. 예시처럼 먼저 각 경로 내에 있는 블록별로 프로그램을 기호 실행하여 각 변수의 상태를 업데이트한다. 기호 형태로 값을 전파하는 과정에서 단순히 명령어를 따라 해당 변수만 업데이트할 뿐만 아니라 포인터 에일리어싱 분석도 동시에 실행한다. 해당 명령어를 실행하므로 동일한 메모리 주소를 가리키는 변수가 존재하면 해당 값도 같이 업데이트한다.

Fig.8.에서는 t5, t4, t2 명령어를 통해서 'deref(bp-0xc)' 와 같은 식을 만드는 과정을 보여준다. 순차적으로 각 변수값과 동일한 의미를 가진 값으로 새로운 식을 만든다. 마지막 if 문의 조건식을 t8에서 'deref(bp-0xc)<0x0'으로 더욱 정확하게 변경하였다. 레지스터, 상수, 메모리 참조 이외의 모든 임시변수는 기본 블록 내에서만 생성되는 변수이다. 따라서 데이터 흐름 분석이 끝난 이후 제거한다.

블록 내 데이터를 기호형태로 상태 분석을 한 뒤 Fig.9.과 같이 실행경로에서 순서대로 블록 간 상태 분석해야 한다. 프로시저 간 분석 경우도 고려하기 때문에 아래와 같은 4가지 경우가 존재한다.

1) 같은 함수 내에 실행순서에 따라 두 블록 상태 업데이트: 같은 함수인 경우 데이터 분석은 동일한 프로그램 상태에서 수행하기 때문에 스택 프레임 포인터 bp는 항상 동일하고 단순히 업데이트한다.

2) 호출자 함수에서 피호출자 함수로 넘어가는 경우: 호출자 함수의 상태는 따로 저장하고 함수 파라미터를 의미하는 레지스터의 값만 피호출자 함수에 전달한다. 그러나, 피호출자 함수에서는 파라미터로 받은 호출자 함수에서 사용한 기호와 충돌이 일어날 수 있어 이와 같은 경우 피호출자 함수에서 사용하는 기호는 모두 '*'를 추가하여 호출자 함수에서 사용하는 기호와 구분한다.

3) 피호출자 함수에서 호출자 함수로 돌아오는 경우: 아키텍처 호출 규약에 따라 리턴 값을 의미하는 레지스터 값이 있는 지 확인한다. 리턴변수가 있으면 피호출자 함수의 다른 변수들은 모두 지우고 리턴변수만 다음에 수행할 호출자 함수의 블록에 전달한다. 피호출자 함수에서 반환한 값을 받기 전에 call site 에서 저장한 호출하기 전의 상태를 회복한 후 계속 실행 순서에 따라 분석을 수행한다.

('func_off_blkid', [('goodB2G', '0x27', '0x40141f')]) **('deref(bp-0xc)', ['0x0'])** ('Jmp_Direct_from_0x401426', [('0x401426', 'NoCond', {})])	('func_off_blkid', [('goodB2G', '0x5a', '0x401452')]) ('cc_op', ['0x7']) ('cc_dep1', ['deref(bp-0xc)']) ('cc_dep2', ['0x0']) ('Jmp_Cond_from_0x401456', [('0x401428', **'deref(bp-0xc)<=0x0**', {}), ('0x401458', **'!deref(bp-0xc)<=0x0**', {})])
('func_off_blkid', [('goodB2G', '0x5a', '0x401452')], [('goodB2G', '0x5a', '0x401452')]) **('deref(bp-0xc)', ['0x0'])** ('Jmp_Direct_from_0x401426', [('0x401426', 'NoCond', {})]) ('cc_op', ['0x7']) ('cc_dep1', ['0x0']) ('cc_dep2', ['0x0']) ('Jmp_Cond_from_0x401456', [('0x401428', **'True**', {}), ('0x401458', **'Dead**', {})])	

Fig. 9. data set merge between block 0x40141f and block 0x401452

4) 프로시저 내 분석만 하는 경우: 프로시저 내 분석이기 때문에 다른 함수를 호출하지 않는다. 함수 호출에 파라미터로 사용된 포인터, 호출하는 함수에서 반환한 값이 모두 기존 기호표현식과 구분되어야 하며 소스로 표기하여 분석을 수행한다.

4.4 한계점 분석

본 논문에서 연구 및 개발한 도구는 라이브러리와 대형 프로그램의 일부 함수를 대상으로 널 포인터 역참조 취약점을 빠르게 탐지하기 위한 도구다. 해당 문제를 풀기 위한 휴리스틱 기반 방법이며 기호실행처럼 프로그램 상태를 수식으로 표현하고 수식풀이를 통해 해당 경로에 도달할 수 있다는 증명과정이 포함되어 있지 않다. 두 번째로, 논문에서 정의한 취약점 탐지 규칙은 엄격한 규칙이기 때문에 모든 취약점을 탐지해 낼 수 있지만, 외부에서 받은 포인터와 메모리 상태를 모르기 때문에 일부 휴리스틱을 사용하였을 때, 오탐이 줄어들었지만 여전히 존재한다. 세 번째로, 프로시저 간 분석 경우, 재귀 함수 호출(recursive call) 과 같이 반복 실행하는 코드에 대해 일정 횟수를 넘으면 분석 중지를 시켜 완전한 분석으로 보기 힘들다.

V. 실 험

본 논문에서 제안하는 방법의 성능을 평가하기 위해 NIST에서 제공하는 Juliet Test Suite C/C++를 사용하였다[17]. Juliet Test Suite는 다양한 취약점 CWE 유형의 데이터를 제공하고 코드 중에 자주 발생하는 취약점 코드 패턴이 모두 포함된 데이터 셋이다. 본 연구에서 널 포인터 역참조 취약점 유형의 프로그램을 실험대상으로 실험하였다.

실험 데이터 셋에 포함된 테스트케이스(testcase) 유형은 Table 2에서 볼 수 있다. Char, class, int, int64, long, struct 유형의 포인터에서 일어난 취약점, if문으로 인한 취약점, Null 포인터 인지 확인한 후 참조하는 경우, 확인하기 전 참조와 같이 취약점 발생이 가능한 여러 가지 상황에 대한 코드로 실험 셋을 구성하였다. 해당 데이터 셋으로 통해 본 논문에서 개발한 도구의 오탐 및 미탐을 기존 도구와 비교하여 성능을 측정한다.

Table 2.에서 Bap_toolkit과 본 논문에서 제안한 도구의 비교 결과이다. 각 testcase 중에도 제어흐름, 데이터 흐름으로 인해 취약점이 발생하는 여러 가지 가능성을 포함한 코드로 실험을 하였다. 본 논문에서 제안한 도구는 취약점일 경우 놓치지 않았다. 그러나 안전한 코드를 취약하다고 판단하는 오판이 존재하였는데, 코드를 분석한 결과 모두 전역변수와 관련되어 있었다. Angr에서는 역어셈블리 했을 때 전역변수를 'lea rax, [rip+offset]' 형태로 먼저 전역변수의 주소를 읽어온 다음 값을 읽어 온다. 본 논문에서 제안하는 방법은 분석한 함수의 스택 내부에 있는 변수만 찾을 수 있고 그 외의 메모리 영역에 저장된 값을 읽어오지 못 하는 한계점이 존재하여 이러한 경우 모두 취약하다고 판단을 하였다.

Table 2. Summary of comparison for the proposed approach vulnerability detection in SARD testset. The FN means the detection tool missed the vulnerability code, and the FP means the tool take the vulnerable code as safe.

# Testcase	# Language	# Program	# Functions	# Known Vulns	Proposed Approach/Bap_toolkit		
					#FPR	#FP	#FN
CWE_476_binary_if	C	18	49	18	0%/0%	0/0	0/0
CWE_476_char	C	42	141	55	15%/5%	8/3	0/39
CWE_476_class	C++	42	141	93	0%/0%	0/0	0/75
CWE_476_deref_after_check	C	18	49	18	0%/0%	0/0	0/18
CWE_476_int	C	42	141	56	14%/0%	8/0	0/40
CWE_476_int64	C	42	141	56	14%/0%	8/0	0/40
CWE_476_long	C	42	141	56	14%/0%	8/0	0/40
CWE_476_null_check_after_deref	C	18	49	45	0%/0%	0/0	0/49
CWE_476_struct	C	42	141	56	14%/0%	8/0	0/40
SUM	C/C++	306	993	453	9%/1%	40/3	0/341

Table 3. Summary of comparison for the proposed approach vulnerability detection in real word binaries. 'Y', 'F', 'N' means the detection result are 'Vulnerable', 'Failed' and 'Safe'

# CVE IDs	# Program	# Version	# Vulnerable Function	# Proposed Approach/Bap_toolkit
CVE-2016-1865	XNU Kernel	3248.50.21	_task_get_assignment	Y/F
		4570.20.62(patched)		N/F
CVE-2017-5668	Libpurple	3.4.1	prplcb_xfer_new_send_cb	Y/F
		3.6.0(patched)		N/F
CVE-2020-6611	LibreDWG	0.93.2564	get_next_owned_entity	Y/N
		0.3.2582(patched)		N/N
CVE-2020-7105	Redis	0.14.0	callbackValDup	Y/N
		1.0.0(patched)		N/N
CVE-2020-14397	LibVNCServer	0.9.12	sraSpanInsertAfter	Y/N
		0.9.13(patched)		N/N

Bap_toolkit에서는 기호실행 방법으로 기호로 프로그램 실행과정을 시뮬레이션을 수행하면서 메모리 상태를 분석 및 기록한다. 따라서 프로그램 시작한 코드부터 분석을 해야 코드 실행으로 인한 프로그램의 상태(변수, 메모리, 포인터 등)의 변화를 정확하게 분석을 할 수 있다. 그러나 검사할 대상이 라이브러리나 프로그램 중 일부 함수인 경우 외부에서 프로그램 상태 정보 없이 포인터를 받았을 때 분석하기 어렵다. Bap_toolkit에서 이를 랜덤 값이 생성하는 방법으로 일단 시뮬레이션을 시작할 수 있게 처리를 하였다. 따라서 탐지하는 과정에서 일어난 FP모두 해당 문제로 인해서 발생한 문제이다. 자동 테스트 말고 수동으로 해당 포인터를 NULL로 지정하여 실행할 수도 있다.

Table 3.는 실제 상용되고 있는 라이브러리나 대형 프로그램에서 발견된 취약점 탐지 결과다. 취약한 함수 버전과 패치된 버전 코드를 모두 테스트하여 취약점 검사의 유효성을 비교하였다. 실험 결과에 따르면 본 방법에서 제안한 방법으로는 해당 테스트케이스에서는 모두 성공적으로 취약한 코드와 안전한 코드를 구분하였다. Bap_toolkit 같은 경우는 XNU Kernel과 Libpurple에서는 모두 실행 실패하였다. 나머지 3개인 경우는 실행을 성공하였지만 탐지결과 모두 안전하다고 판단을 내렸다. 이는 앞서 서술한 것 과 같이 임의 값으로 테스트하여서 취약한 코드를 못 찾은 것이다. 그러나 Bap_toolkit과 같은 방법은 일반적인 완전한 프로그램을 분석하고 취약점을 탐지하는 문제에서는 강력한 분석도구다.

VI. 결 론

퍼징과 같은 동적 탐지 기법은 탐지된 결과가 정확하고 프로그램에 존재하는 취약점을 찾을 때 유용하다. 특히 패턴이 알려지지 않은 취약점을 찾을 때 매우 유용한 방법이다. 그러나 일부 기능을 탐지할 때 또는 특정한 패턴의 취약한 코드를 찾는 경우에는 정적분석이 더욱 적절한 문제 해결 방법이다.

Bap과 같은 도구는 기호실행 방법으로 취약점 탐지에 정적분석 특성 및 동적 분석의 장점을 잘 결합한 좋은 방법이다. 그러나 실험 결과를 분석해보자면 모든 환경을 잘 갖춰져 있을 때는 매우 강력한 도구이나, 사용 환경에 대한 제한점이 존재한다. 프로그램이 매우 복잡하고 클 경우, 일부 프로그램 기능만 테스트했을 경우 한계점이 존재한다.

본 논문에서는 라이브러리나 대형 프로그램 중 일부 함수에서 널 포인터 역참조 취약점을 탐지하기 위한 도구의 연구를 먼저 수행하여 멀티 바이너리에서 함수 내 및 함수 간 데이터 의존성 분석 기능을 연구 및 개발하였다. 이를 바탕으로 취약점을 탐지하는 도구 개발하였다. 본 논문에서 제안한 도구는 사용 환경에 대한 요구가 없고 모든 함수로부터 분석이 가능하며 널 포인터 역참조 취약점뿐만 아니라 데이터 흐름 분석을 이용한 다른 취약점 탐지에도 쉽게 적용할 수 있다. 그러나 완전하지 않고 휴리스틱한 방법이기 때문에 오탐이 존재하여 특정한 문제 범위에서는 빠르고 유용한 도구이다.

References

[1] CWE Top 25 2021, "CWEtop25 2021" https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, Dec. 2022.

[2] Thompson, Ken. "Reflections on trusting trust," Communications of the ACM, vol.27, no. 8, pp. 761-763, Aug. 1984

[3] Shoshitaishvili, Yan, et al., "Sok:(state of) the art of war: Offensive techniques in binary analysis," Proceedings of the 2016 IEEE symposium on security and privacy (S&P), pp. 138-157, May. 2016

[4] Li, Yuekang, et al., "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 533-544, Aug. 2019

[5] Peng, Hui, Yan Shoshitaishvili, and Mathias Payer, "T-Fuzz: fuzzing by program transformation," Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P), pp. 697-710, May. 2018

[6] Chen, Chen, et al., "A systematic review of fuzzing techniques," Journal of Computers & Security, vol.75, pp. 118-137, 2018

[7] Gan, Shuitao, et al., "Collafl: Path sensitive fuzzing," Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P), pp. 679-696, May. 2018

[8] Fioraldi, Andrea, et al., "{AFL++}: Combining Incremental Steps of Fuzzing Research," Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), pp. 10-10, Aug. 2020

[9] A. Filieri, C. S. Păsăreanu, andW.Visser, "Reliability analysis insymbolic pathfinder," Proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 622 - 631, May. 2013

[10] J. H. Siddiqui and S. Khurshid, "Staged symbolic execution," Proceedings of the 27th Annual ACM Symposium on Applied Computng, pp. 1339 - 1346, Mar. 2012

[11] Wang, Tielei, et al., "IntScope: Automatically Detecting Integer Overflow Vulnerabilityf in X86 Binary Using Symbolic Execution." Proceedings of the Network and Distributed System Security Symposium (NDSS), Feb. 2009

[12] Davidson, Drew, et al. "{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution," Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13), pp. 463-478, Aug. 2013

[13] Yao, Fan, et al., "Statsym: vulnerable path discovery through statistics-guided symbolic execution," Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 109-120, Jun. 2017

[14] Li, Hongzhe, et al., "Software vulnerability detection using backward trace analysis and symbolic execution," Proceedings of the International Conference on Availability, Reliability and Security, pp. 446-454, Sep. 2013

[15] Luo, Lannan, et al., "System service call-oriented symbolic execution of android framework with applications

to vulnerability discovery and exploit generation," Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pp. 225-238, Jun. 2017

[16] Chen, Ting, et al., "State of the art: Dynamic symbolic execution for automated test generation," Journal of Future Generation Computer Systems, vol.29, no. 7, pp. 1758-1773, 2013

[17] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley., "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," Proceedings of the IEEE symposium on Security and privacy (S&P), pp. 317-331, May. 2010

[18] Ruaro, Nicola, et al., "SyML: Guiding symbolic execution toward vulnerable states through pattern learning," Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 456-468, Oct. 2021

[19] Cheng, Kai, et al., "DTaint: detecting the taint-style vulnerability in embedded device firmware," Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 430-441, Jun. 2018

[20] Wang, Tielei, et al., "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," Proceedings of the IEEE Symposium on Security and Privacy (S&P), pp. 497-512, May. 2010

[21] CVE-2017-5668, "CVE-2017-5668", https://nvd.nist.gov/vuln/detail/CVE-2017-5668, Dec. 2022.

[22] libpurple, "libpurple", https://developer.pidgin.im/wiki/WhatIsLibpurple, Dec. 2022.

[23] Brumley, David, et al., "BAP: A binary analysis platform," Proceedings of the International Conference on Computer Aided Verification, pp. 463-469, Jul. 2011

[24] IDA Pro, "IDA Pro", https://hex-rays.com/ida-pro/, Dec. 2022.

[25] AFL, "AFL Fuzzer", https://github.com/go ogle/AFL, Dec. 2022.

[26] Nethercote, Nicholas, and Julian Seward., "Valgrind: a framework for heavyweight dynamic binary instrumentation," Jounral of the ACM Sigplan notices, vol.42, no. 6, pp. 89-100, 2007

[27] Gotovchits, Ivan, Rijnard Van Tonder, and David Brumley, "Saluki: finding taint-style vulnerabilities with static property checking," Proceedings of the NDSS Workshop on Binary Analysis Research, Jul. 2018

[28] Ma, Sen, et al., "Practical null pointer dereference detection via value-dependence analysis," Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 70-77, Nov. 2015

〈저 자 소 개〉

김 문 회 (Wenhui Jin) 학생회원
2013년 9월: HEILONGJIANG UNIVERSITY 소프트웨어공학 학사
2016년 3월~현재: 한양대학교 컴퓨터공학과 석박사통합과정
〈관심분야〉 정보보호, 전자공학, 통신공학

오 희 국 (Heekuck Oh) 종신회원
1982년: 한양대학교 전자공학과 졸업
1989년: 아이오와주립대학 전자계산학과 석사
1992년: 아이오와주립대학 전자계산학과 박사
1993년~1994년: 한국전자통신연구원 선임연구원
1995년 3월~현재: 한양대학교 ERICA 소프트웨어융합대학 교수
〈관심분야〉 암호기술응용, 시스템보안