소프트웨어 보안약점 분석을 위한 바이너리 코드-중간언어 변환기에 관한 연구

이태규*, 임정호*, 백도우*, 손윤식**, 정준호***, 고광만****, 오세만**
*고려대학교 사이버국방학과
**동국대학교 컴퓨터공학과
***동국대학교 경주캠퍼스 전자성거래 연구소
*****상지대학교 컴퓨터정보공학부
e-mail: smoh@dongguk.edu

A Study of The Binary Code to Intermediate Language Translator for Analysis of Software Weakness

Do-Woo Baik*, Tae-Gue Lee**, Jung-Ho Lim*, Yunsik Son**, Junho Jeong***, Kwangman Ko****, Seman Oh**
*Dept of Cyber Defense, Korea University
**Dept of Computer Engineering, Dongguk University
***Electronic Commerce Institute, Dongguk University Gyeongju Campus
****Dept. of Computer and Information Engineering, SangJi University

요 약

오늘날 사회 전반적인 부분에서 소프트웨어의 비중은 지속적으로 증가하고 있다. 또한 소프트웨어는 점차 대규모화되고 있고 동시에 개인의 중요한 정보 등을 다루는 경우도 매우 늘어나고 있기에 소프트웨어의 보안성 검증은 매주 중요한 문제이다. 그러나 소스코드가 존재하지 않는 라이브러리의 경우 보안성 검증은 매우 어려운 문제로, 이를 해결하기 위해 바이너리 내에 존재하는 보안약점을 검사하기 위한 기술의 개발이 매우 요구되는 상황이며, 이를 위해 중간언어를 활용하여 보안약점을 분석하는 기술이 활발히 논의되고 있다. 본 논문에서는 바이너리 코드내에 존재하는 보안약점을 효과적으로 분석하기 위해서 바이너리 코드로부터 보안약점 분석에 효과적인 중간언어로 변환하는 시스템을 제안한다.

1. 서론

오늘날 사회 전반적인 부분에서 소프트웨어가 차지하는 비중은 지속적으로 증가하고 있다. 또한 이러한 소프트웨어는 점차 대규모화되고 있고 동시에 개인의 중요한 정보 등을 다루는 경우도 매우 늘어나고 있기에 소프트웨어의 보안성 검증은 매주 중요한 문제이다.[1]

특히, 최근의 소프트웨어 개발은 오픈소스 및 서드파티 라이브러리를 활용하는 경우가 급증하고 있다.[2] 반면, 서드파티 라이브러리를 사용하는 경우, 개발과정에서 보안성 분석 및 테스트가 이루어지는 경우는 50%이하로 조사되어 있으며[3] 이러한 라이브러리의 사용으로 인한 보안 사고는 HeartBleed[4], ShellShock[5], POODLE[6], DROWN[7] 등 지속적으로 증가하고 있다.

소프트웨어 개발에 사용된 라이브러리 내에, 알려진 기능 이외의 숨겨진 악의적인 기능이 있거나 프로그램 실행 중에 발생할 수 있는 보안약점이 있을 경우, 심각한 결함을 야기할 수 있으므로, 이 경우 서드파티 라이브러리에 대한 보안성 검증은 반드시 필요한 작업이다.

그러나, 소스코드가 존재하지 않는 라이브러리의 경우 일반적인 보안약점 분석에 사용되는 소스코드 상의 자료형이나 변수 등이 모두 사라지기에 보안성 검증은 매우 어려운 문제이다. 최근 라이브러리와 같은 바이너리 파일을 대상으로 보안약점을 분석하는 다양한 연구가 진행되었는데 그 중에서도 최근의 주요 연구흐름은 바이너리 코드를 중간언어로 변화하고 이 중간언어를 기반으로 보안약점을 분석하는 연구가 많이 진행되었다.[8-12]

이와 같은 보안약점 분석이 효과적으로 이루어지기 위해서는 보안약점 분석에 효과적인 중간언어가 선정되어야하며 뿐만 아니라 바이너리 코드로부터 중간언어로 변환도 잘 이루어져야 할 것이다. 본 연구는 바이너리 코드 내에 존재하는 보안약점을 효과적으로 분석하기 위해서 바이너리 코드로부터 보안약점 분석에 효과적인 중간언어로 변환하는 시스템을 제안한다.

본 논문의 구성은 다음과 같다. 2 장에서 관련연구를 살펴보고 3 장에서 제안하는 시스템을 소개한다. 4장에서 실험 결과를 분석하며 마지막 장에서 결론과 향후연구를 제시한다.

"본 연구는 방위사업청과 국방과학연구소 (계약번호 UD160035ED)의 연구비 지원에 의한 연구 결과임"

2. 관련 연구

바이너리 기반 보안약점 분석기법은 크게 역공학 기반, 바이너리 직접분석, 중간언어 변환 분석의 세 가지 종류로 분류 할 수 있으며 각각의 기술은 장단점이 있다.

역공학 기반의 분석 기법은 전문가가 바이너리 파일을 분석할 경우 기존에 없던 보안약점을 탐지할 수 있으며 기존에 존재하지 않은 보안약점을 찾아낼 수 있는 장점이 있으나 사용자의 숙련도에 따라서 보안약점 탐지 결과가 상이 할 수 있고, 데이터의 수가 증가하거나 안티 디버깅 방법들이 존재할 경우 데이터 분석이 어려운 단점이 있다.[13-15]

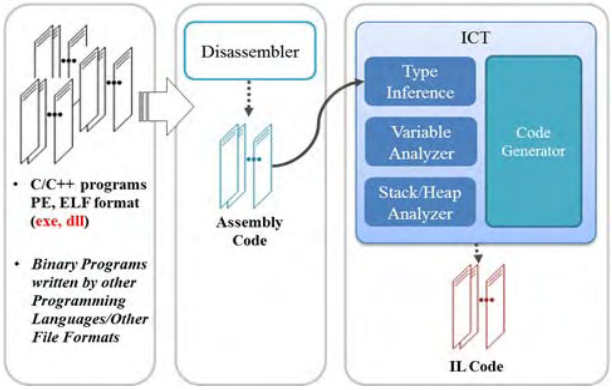
바이너리 직접 분석 방법은 바이너리 파일을 대상으로 자동화 도구를 사용하여 분석할 수 있고 기존에 존재하는 디스어셈블러를 통하여 보안약점 데이터를 확인할 수 있는 장점이 있다. 그러나 운영체제 및 CPU에 종속적이고 도구의 확장성이 낮고 변형된 형태의 보안약점의 경우 탐지율이 떨어지는 문제가 있다.[16,17]

중간언어 변환 분석 기법은 플랫폼 및 언어에 독립적일 수 있으나 이를 위한 중간언어를 설계하는 과정이 복잡하고 구현과정에서 시간이 오래 소요되는 단점이 있다.[8,9] 그럼에도 불구하고 목적하는 보안약점 분석에 효과적인 중간언어를 설계하고 구현이 된다면 기존에 존재하는 보안약점을 대상으로 자동화 도구를 통해 분석할 수 있는 큰 장점이 있다.

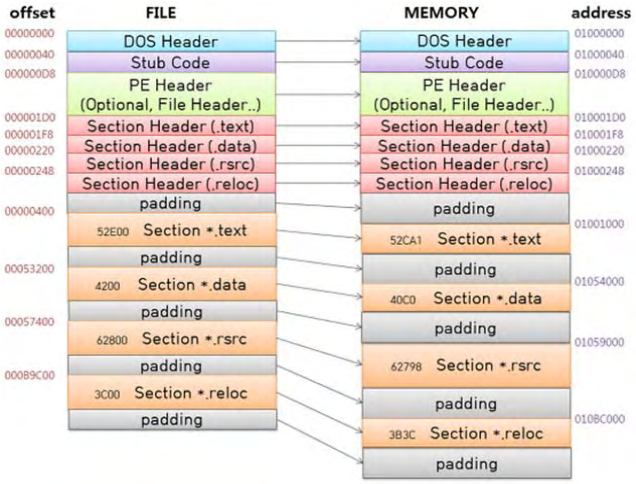
3. 바이너리코드-중간언어 변환기

제안 시스템은 바이너리 코드내에 존재하는 보안약점을 효과적으로 분석하기 위해서 바이너리 코드로부터 보안약점 분석에 효과적인 중간언어로 변환하는 시스템으로 (그림 1)과 같다.

그 과정은 크게 세단계로 표현될 수 있다. 첫 번째 단계에서는 C/C++로 작성된 바이너리 파일을 분석하여 해당 파일의 작성 언어 및 형태에 대한 분석을 통해 분류하는 것이다. 동일한 소스코드라고 할지라도 컴파일된 컴파일러와 대상 기계에 따라 다른 형태의 바이너리 코드가 생성되기에 사전 분류가 요구된다. 따라서 제안 시스템은 C/C++로 작성된 PE 포맷의 x86/64 bit 바이너리 코드를 대상으로 분석을 수행하며 이외의 대상에 대해서는 오류 처리된다.



(그림 1) 제안 시스템 구성도



(그림 2) PE 구조 (RAW, RVA)

두 번째 단계는 바이너리 코드의 분석을 통해 파싱하여 각 함수를 추출하고 추출된 함수에서 일반 블록을 구성하여 어셈블리 코드로 변환하며 이를 바탕으로 제어 흐름 그래프(Control Flow Graph/CFG)를 추출하는 것이다.

(그림 2)는 변환기의 입력으로 사용되는 PE 포맷구조에서 사용되는 두 가지 메모리 개념인 RAW, RVA(Related Virtual Address)를 보여준다. RAW는 파일에서 사용되는 오프셋이며 이것은 실제 프로그램이 메모리에 로드될 때는 재배치됨에 따라 오프셋이 달라지며 이를 RVA 라 한다. 그러므로 이를 고려하여 올바르게 계산해야 추출할 함수의 오프셋을 역으로 계산하여 파일의 어느 위치에서 추출할지 알아낼 수 있다.

바이너리 코드에서 분석되는 주요 섹션은 .rdata 영역이며 rdata 영역 내에는 IMAGE_EXPORT_DIRECTORY 구조체가 있고 이 구조체는 해당 코드가 export하는 함수가 몇 개인지, 이름이 무엇인지, 어떤 offset(RVA)에 저장되는지에 대한 정보를 담고 있다. 따라서 이 정보를 이용하여 바이너리 코드에 존재하는 함수를 구별·추출할 수 있다.

CFG는 프로그램을 실행하면서 통과 할 수 있는 모든 경로를 그래프 표기법을 사용하여 표현한 것으로 어셈블리 명령어와 제안 시스템에서 사용하는 SIL 명령어의 근본적인 차이를 해결하기 위해서 어셈블리 명령어를 통해 CFG를 만드는 것이 중요하다.

CFG는 프로그램의 제어 흐름 분석을 시작으로 전체적인 프로그램의 구성을 파악할 수 있고 기본 블록을 스캔하여 프로그램을 보다 효율적으로 구축 할 수 있기 때문에 중요하다. 예를 들어 Reachability는 최적화에 유용한 그래프 속성이다. 서브 그래프가 엔트리 블록을 포함하는 서브 그래프로부터 연결되어 있지 않으면, 그 서브 그래프는 실행 중에 도달 할 수 없으므로 도달 할 수 없는 코드라고 생각할 수 있다. 따라서 정상적인 조건으로 생각해 볼 때, 이 부분을 제거 할 수 있다.

제안 시스템의 CFG에서는 각 함수의 호출 관계는 나타내지 않고 조건에 따른 분기문(branch)이 나타날 때 흐름을

분리하여 Graph를 나타낸다. 예를 들면, jnz 0x1234 라는 구문에 0x1234라는 주소로 jmp하거나 다음 명령어로 넘어가거나 두 가지로 경우의 수가 나타난다.

마지막 단계는 어셈블리로부터 도출된 CFG를 바탕으로 자료형에 대한 정보가 포함된 명령어로 효과적인 보안약점 분석이 가능한 SIL 중간언어로 변환하는 과정이다.

제안하는 시스템은 바이너리 코드를 입력으로 받아 그것을 SIL 중간언어로 변환하는 것은 몇 가지 어려운 점이 있다. 첫 째, 너무 많은 어셈블리 명령어들이 있는 것이다. 둘 째, x86/64 명령어들은 기본적으로 파이프라인 구조 바탕의 명령어로 각 파이프라인마다 레지스터들이 다를 수 있다. 셋 째, 하나의 어셈블리 명령어가 너무 많은 SIL 명령어로 대체 되는 것이다. 따라서, 그 중 SIL 중간 언어로 변환을 위해서 필수적으로 해결되어야 하는 것은 첫 번째와 두 번째 문제점이다.

첫 번째 문제를 해결하기 위해서는 어셈블리 명령어 집합을 줄여야한다. 즉, 일반화된 어셈블리 명령어 집합을 만들어야 한다. 구체적인 방법은 다음과 같다. 어셈블리 명령어는 CISC (Complex Instruction Set Computer) 명령어이고 SIL은 RISC (Reduced Instruction Set Computer) 명령어이다. 이것을 더욱 쉽게 변환하려면, 어셈블리 명령어를 RISC 명령어들로 바꾸는 작업이 필요하다. 따라서 먼저 복합연산을 수행하는 어셈블리 명령어들을 단순 명령어 형태의 어셈블리 RISC 명령어로 치환해주어야 한다. 또한 EFLAGS, 레지스터의 정보를 저장할 수 있는 독립적인 스택 공간을 할당하는 것으로 이 문제를 해결할 수 있다. 그래서 우리는 RISC 명령어 중 flag를 사용하지 않는 명령어와 조건 분기문 명령어를 구현하였다.

두 번째 문제점에서 파이프라인은 컴파일 상에서 짧아졌으면 문제가 되지만, 앞선 문제해결을 통해 SIL 명령어의 변환은 오히려 문장이 길어지기 때문에 이것이 문제가 되지 않는다. 그에 따라 최종적으로 구현해야할 일반 명령어 집합을 정리하면 <표 1>과 같다.

<표 1> 일반화된 어셈블리 명령어 집합

스택관리	nop	pop	push	
논리&산술 연산	mov	add	sub	xor
	or	and	xchg	shl
	sal	shr	sar	rol
	ror	not	mul	imul
	div	idiv	neg	inc
	dec	cmp	test	
제어흐름	jmp	ja	jnbe	jae
	jnb	jb	jnae	jbe
	jna	jg	jnle	jge
	jnl	jl	jnge	jle
	jge	jz	jnz	ret
	call			
문자열 처리 명령어	rep	movs	lods	stos

<표 2> 레지스터와 스택주소의 사상

레지스터이름(x86기준)	stack의 (base, offset)
rax	(0, 0)
rbx	(0, 8)
rcx	(0, 16)
rdx	(0, 24)
rsp	(0, 32)
rbp	(0, 40)
rsi	(0, 48)
rdi	(0, 56)
rip	(0, 64)

또한 앞서 언급한바와 같이 x86/64에는 cpu 안에 레지스터가 존재하지만 변환하고자 하는 SIL 언어는 스택 기반의 언어로 레지스터가 존재하지 않는다. 따라서 레지스터의 역할을 할 수 있는 것이 필요한데 우리는 이것을 레지스터와 스택의 특정영역을 일대일 대응하여 할당해 주었다. 즉, 레지스터도 일종의 스택의 연속이라고 생각하며 변환하는 것이다. 따라서 본 프로그램 에서는 <표 2> 와 같이 레지스터를 스택의 영역에 미리 할당해주었다.

또한 중간언어인 SIL은 다루는 데이터에 따라 명령어가 달라지므로 바이너리 파일로부터 효과적인 데이터에 대한 자료형의 추론이 필요하다. 제안 시스템에서는 지역 변수 중 정수형 자료형을 중심으로 char, short, int 변수와 int 배열 타입을 추론할 수 있다. 찾는 패턴은 다음과 같이 [ebp ± 상수], [esp ± 상수], [ebp + 레지스터 * 상수 - 상수]이다.

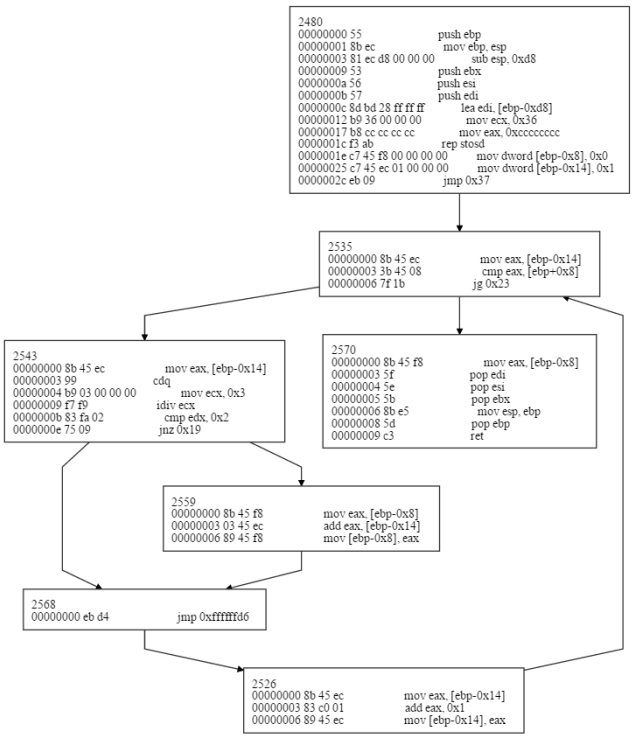
4. 실험결과 분석

본 연구에서는 다양한 테스트 케이스를 활용하여 변환을 테스트 하였다. (그림 3)은 그 중 하나이며 이를 바이너리 파일로 변환하여 SIL 코드로의 변환을 테스트 하였다.

(그림 4)는 두 번째 단계인 바이너리 코드를 분석을 통해 파싱하여 각 함수를 추출하고 추출된 함수에서 Normal Block을 구성하여 이를 바탕으로 CFG를 추출한 결과이다.

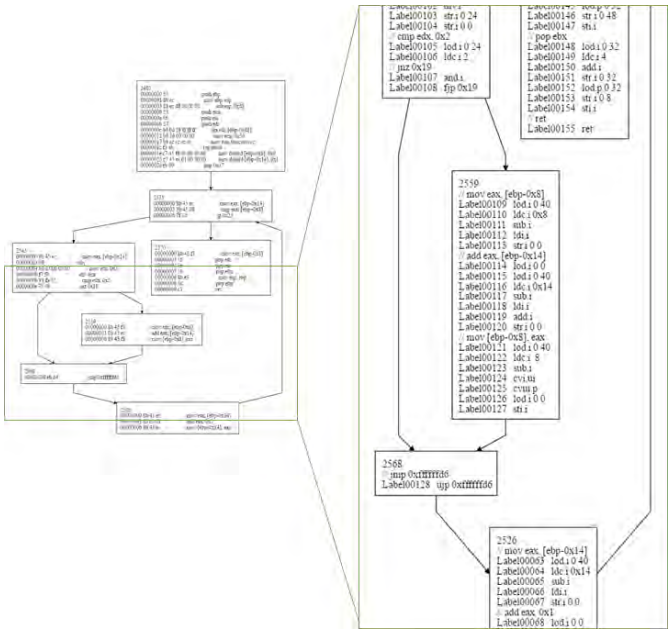
```
DLL_EXPORTING_API int DLL6(int n)
{
    int sum = 0;
    int i;
    for (i = 1; i <= n; i++)
    {
        if (i % 3 == 2)
            sum += i;
    }
    return sum;
}
```

(그림 3) 분석 대상 프로그램



(그림 4) 바이너리코드로부터 추출된 CFG

또한 (그림 5)는 바이너리 코드로부터 작성된 CFG를 바탕으로 SIL 코드 CFG로 변환한 결과이다. (그림 4)의 2559, 2568, 2526 블록 등이 SIL로 확장 변환된 것을 확인할 수 있으며 제어흐름은 변하지 않는 것을 확인할 수 있다.



(그림 5) SIL 코드 CFG

5. 결론

우리는 본 논문을 통해 바이너리 코드로부터 보안약점 분석에 효과적인 중간언어 SIL로 변환하기 위한 방법을 제안하였다. 몇 가지 어려운 점이 있었지만 논문에서 제안한 방식을 통해 해당 문제점을 해결하였다. 향후연구로 확인되었던 문제점 중 하나인 변환된 SIL 언어가 너무 길어지는 문제를 딥러닝 기법을 활용하여 축약하고 자료형 및 변수 추론의 성능을 높이고자 한다.

참고문헌

[1] 류성일, “한국 소프트웨어 산업의 현황 및 제언,” kt 경제경영연구소, 2014. 06, 11
[2] Find Defects in Third-Party Code, http://www.grammatech.com/products/binary-analysis
[3] Synopsys, Study Reveals Less Than Fifty Percent of Third Party Code Is Tested for Quality and Security in Development, http://www.coverity.com/press-releases/report-reveals-less-than-fifty-percent-of-third-party-code-is-tested-for-quality-and-security-in-development/
[4] codenomicon, The Heartbleed Bug, http://heartbleed.com/
[5] US-CERT, GNU Bourne-Again Shell ‘ShellShock’ Vulnerability, https://www.us-cert.gov/ncas/alerts/TA14-268A
[6] Bodo Möller, This POODLE Bites : Expoiting The SSL 3.0 FallBack, google
[7] Nimrod Aviram, DROWN : Breaking TLS using SSL v2, https://drownattack.com
[8] Song, Dawn, et al. "BitBlaze: A new approach to computer security via binary analysis." Information systems security. Springer Berlin Heidelberg, 2008. 1-25.
[9] GRAMMATECH, “Eliminating Vulnerabilities in Third-party Code with Binary Analysis”, WhitePaper http://www.grammatech.com/products/codesonar
[10] Necula, George C., et al. "CIL: Intermediate language and tools for analysis and transformation of C programs." Compiler Construction. Springer Berlin Heidelberg, 2002.
[11] Dullien, Thomas, and Sebastian Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis." Proceeding of CanSecWest (2009).
[12] Cesare, Silvio, and Yang Xiang. "Wire-A Formal Intermediate Language for Binary Analysis." Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on. IEEE, 2012.
[13] WinDbg, http://www.windbg.org/
[14] OllyDbg, http://www.ollydbg.de/
[15] Pro, https://www.hex-rays.com/products/ida/
[16] C A T . N E T , https://www.microsoft.com/en-us/download/details.aspx?id=19968
[17] CodeSurfer, http://www.grammatech.com/products/codesurfer