어셈블리 언어 수준에서의 소스코드 보안취약점 점검방법에 관한 연구

박현미*, 이병권*, 박정현**, 이형봉***
한국정보보호진흥원*, 호남대학교***

The Study of Checking Source Code Vulnerability on the assembly language level

HyunMee Park, ByungKwon Lee, JeongHyun Park, HyungBong Lee
Korea Information Security Agency, ,Ho-Nam University

요 약

대부분의 해킹 공격은 공격 대상 프로그램의 소스코드 보안취약점에 의해서 발생하지만 프로그램 개발시에 소스코드 보안성에 대해서는 고려되지 않았다. 이러한 문제점으로 인하여 해킹 공격의 근본적인 원인을 해결할 수 없었다.
본 논문에서는 취약점의 원인이 되는 코드를 컴파일시 생성된 어셈블리 코드 수준에서 탐지하는 방법을 제시하고자 한다. 취약한 코드를 컴파일러 수준에서 점검하는 것보다 어셈블리 코드 수준에서 점검하는 것은 어느 정도의 메모리 영역까지 점검할 수 있어 더 정확하다.

Ⅰ. 서론

지난 7월에 급속도로 확산된 서캠 바이러스[7]는 내 컴퓨터의 주소록과 캐쉬에 등록되어 있는 모든 메일주소로 문서를 첨부하여 보내는 바이러스다. 이 바이러스 인하여 많은 기업은 기업의 보안문서가 첨부되어 메일로 보내지는 바람에 보안문서가 노출되어 문제를 일으켰고 메일을 받은 수신자는 끝없이 쌓이는 메일로 인하여 업무에 지장을 받았을 것이다.

사용자의 무지함과 부주의함에 의해 바이러스가 확산되었지만 안티바이러스 제품을 설치하고 운영하였던 곳에서는 이 안티바이러스 제품이 제대로 바이러스를 탐지하고 제거해주지 않아 안심하고 있다가 피해를 입은 것이라 할 수 있다. 안티바이러스 제품이 서캠 바이러스를 제대로 차단하지 못한 것은 탐지를 실패했을 때 디폴트로 처리(failed open)되었기 때문이다.

비단 안티바이러스 제품에서만 이러한 문제가 발생하는 것은 아니다. 침입차단시스템 시장에서 리더격인 CheckPoint사의 FireWall-1은 매년 최소한 4개의 취약점을 발표하고 있다.[7] 보안제품이 이런 상황인데 일반 소프트웨어 제품은 어떠하겠는가?

안전한 코드로 제품을 개발하는 것은 쉽지 않

다. 안전하게 설계하는 것은 더더욱 어렵다. 그래서 본 고에서 안전한 코드로 제품을 개발하는데 도움을 줄 수 있는 코드 취약성 탐지툴을 제안하고자 한다. 특히 여러 가지 탐지방법 중에 개발언어에 독립적인 어셈블리언어 레벨에서의 취약성 탐지 방법에 대하여 제안하고자 한다.

Ⅱ. 관련연구

소스코드상의 오류에 관한 연구는 소프트웨어 품질(Quality)에 관한 연구로 진행되어 왔는데 양질의 소프트웨어를 개발하기 위하여 시스템의 안정성과 신뢰성을 높이며 최소한의 유지보수 노력으로 경제성을 높이는 개발활동으로 다음과 같은 활동이 있다.

- **소프트웨어 품질보증(QA : Quality Assurance)** - 어떠한 소프트웨어 제품이 이미 설정된 요구사항과 일치하는가를 확인하는데 필요한 계획적이면서도 체계적인 작업.
- **소프트웨어 품질관리(QC : Quality Control)** - 주어진 요구를 만족시키는 제품 혹은 서비스의 질을 보존하는데 필요한 제반기법과 활동
- **소프트웨어 시험(Test)** - 오류를 찾아내기 위하여 프로그램을 수행시키는 과정
- **오류수정(Debugging : 디버깅)** - 노출된 오류의 본질을 정확히 진단하고 이를 바르게 고치는 활동

즉 소프트웨어의 품질보증을 위하여 품질관리를 전개시켜야 하고 품질관리는 소프트웨어 시험과정과 오류수정 과정을 거쳐 소프트웨어의 품질을 검사자의 입장에서 재확인하는 것이다. 따라서 요구사항에 합당한 소프트웨어가 개발되었는지, 개발자가 시험환경의 오류를 발견하는 검증(Verification)이나 검사자가 실제 환경에서의 실행시 오류를 발견하는 확인(Validation), 사용자가 소프트웨어의 품질을 공식적으로 확인하는 인증(Certification) 과정에서 소스코드의 오류를 수정하는 연구로 진행되어 왔다.

그러나 인터넷의 급격한 확산과 함께 정보에 대한 신뢰성(Reliability)과 안정성(Safety), 보안성(Security)에 대한 요구가 증가하면서 하나의 독립된 분야인 언어기반 보안과 소스코드 신뢰성 점검 도구 개발로 연구가 진행되고 있다.

1. 안전한 소스코드 검증을 위한 점검툴

안전한 소스코드 검증을 위한 점검툴의 개발은 기존의 소프트웨어 품질에서의 소프트웨어 시험(Test)나 오류수정(Debugging)에서의 사후 보증이 아닌 사전 보증을 목표로 한다. 즉, 소프트웨어를 개발하는 초기 단계에서부터 안전한 소스코드로 개발하도록 유도하여 후에 발생할 수 있는 최소한의 오류까지 제거한다.

현재 국외에서는 소스코드의 안전성을 점검하는 툴의 개발이 활발하게 진행되고 있다. 가장 많이 보고되는 버퍼오버플로우 취약점이나 레이스컨디션, 포맷스트링 등의 취약한 코드를 점검해줄 뿐 아니라 프로그래머가 흔히 범할 수 있는 실수까지 점검해 준다. 특히 이러한 점검 툴들은 취약한 코드를 탐지하는 여러 가지 알고리즘에 따라 점검한다.

벤더	툴/제품	목표	접근방법
@Stake	SLINT	C/C++	Static
Cigital	ITS4	C/C++	Static
U of Va	LCLint	C/C++	Static
Avaya Labs	LibSafe	C/C++	Run-Time
Rational S/W	Purify	C/C++	Run-Time
Wirex	StackGuard	C/C++	Run-Time

표 1 : 소스코드 점검툴

2. 언어기반 보안 접근 방법론[4]

"신뢰할 수 없는 사이트에서 프로그램을 다운받아 사용해야 할 경우 그 프로그램이 악의적이지 않거나 의도하지 않았던 어떤 동작도 하지 않는다고 어떻게 확신할 수 있는가?"라는 질문은 오늘날과 같이 이동 코드(mobile code)가 이슈가 되는 컴퓨팅 환경(computing environment)에서 고려되어야 할 문제점이다.

이러한 문제점에 대한 접근방법을 **언어기반 보안(language-based security)**이라고 하는데 현재 연구되고 있는 접근방법에는 PCC(Proof Carrying Code), TAL(Tightly-typed Assembly Language), ECC(Effient Code Certification)와 Information Flow 등이 있다.

○ Proof Carrying Code : PCC

PPC는 신뢰할 수 없는 코드를 안전한 실행하기 위한 기술로 code producer는 safety rule을 증명하기위해 형식상의 safety proof를 생성하고 code receiver는 프로그램의 안전한 동작을 보증하는 safety rule 집합을 확립한다.

그 동작과정은 다음과 같은데 producer는 소스 프로그램을 컴파일 하여 dataflow 특성이나 type의 정보를 prover에게 제공하여 safety proof를 생성한다. 컴파일하여 생성된 기계어 코드는 VCgen(Verification Condition)에의해 그에 매핑되는 safety theorem를 생성한다. code receiver는 safety theorem이 프로그램과 매핑되는지 safety proof가 safety theorem과 매핑되는지 검사하는데, 매핑되면 그 프로그램이 안전하게 실행되는 것이다.

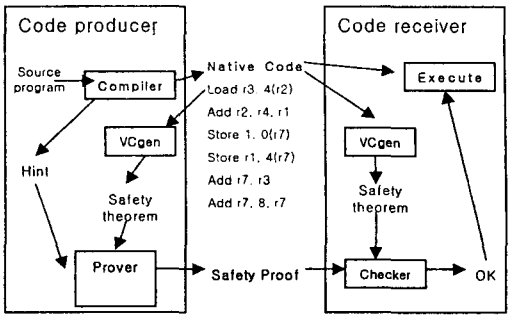


그림 1 : PCC의 동작과정

○ Typed Assembly Language : TAL

TAL은 전통적으로 type을 가지고 있지 않은 어셈블리 언어를 typing annotation이나 memory menagement primitive, typing rule 집합으로 확장한 것으로 이동 코드(mobile code) 어플리케이션이나 확장된 운영체제 커널에서 안전한 코드를 생성하기위한 type-directed 컴파일러의 이상적인 형식이다.

Typing rule은 memory safety와 control flow safety, type safety를 보장하고 typing 구조는 레코드나 구조체, 배열, higher-order와 다형의 함수, 예외, 추상 데이터 타입, subtyping, 모듈을 포함하는 대부분의 소스 프로그램의 특징을 표현할 수 있다.

○ Effient Code Certification : ECC

ECC는 신뢰할 수 없는 곳으로부터 다운로드한 컴파일된 코드를 증명하는 단순하고 효율적인 접근방법으로 control flow safety, memory safety와 stack safety와 같은 코드의 안전성을 포함한다. PCC나 TAL보다 표현력이 부족하지만 ECC의 증명은 생성하고 검증하기가 쉽고 정밀하다. 또한 ECC는 자바의 바이트 코드와 달리 기계어 수

준에서 수행되므로 인터프리터나 컴파일이 필요하지 않아 수행시간의 부하가 걸리지 않는 이점이 있다.

○ Information Flow : IF

또한 언어기반 보안 방법은 서로 신뢰할 수 없는 에이젼트간의 Information Flow를 제어하는데 사용할 수 있다. Information Flow는 보안 정책이 정보 흐름 모델을 기반으로 한다는 것을 제외하고 앞에서 설명한 언어기반 보안 방법과 비슷하다. 이러한 정책은 사용자에의해 고급언어의 주석으로 명시되는데 이것은 한 프로그램이나 프로그램 사이의 정보흐름를 제한한다. 그래서 주석이 있는 프로그램은 컴파일시에 점검된다.

JFlow 컴파일러는 type-checking 메커니즘을 이용하여 Information Flow Safety를 점검하고 명시된 주석을 지운다. Object Code를 다운받아 전달되는 Control Information은 TAL이나 ECC와 비슷한 방법으로 검증된다.

Ⅲ. 연구 내용

여기에서는 기계어 코드 수준에서 코드의 취약점을 점검하기 위하여 먼저 기계어 코드 안전성 점검 기술에 대하여 알아보고 기계어 코드 수준에서 코드의 취약점을 점검하기 위한 방법과 점검 툴을 구현하고자 한다. 단, 본 논문에서 사용한 기계어 코드는 코드 이해의 용이성을 위하여 기계어와 일대일 매핑되는 어셈블리 언어를 사용했음을 미리 알려두는 바이다.

1. 기계어 코드 안전성 점검 기술

기계어 코드를 입력으로 받아 코드에 대한 안전성을 검증하기 위해서는 기계어 코드가 내재하고 있는 정보에 대하여 검증에 필요한 정보로 추출해야 한다. 코드 안전성 점검에 필요한 정보로는 변수나 함수의 타입과 크기, 함수의 호출로 변경되는 정보 등이 있는데 이러한 정보를 추출하기 위한 기술들이 있다.

○ 타입 조사 시스템

기계어 코드에서 레지스터나 메모리 위치는 주로 프로그램이 실행되는 과정에서 단순 타입의 값을 저장하거나 다른 프로그램으로 분기한 경우 다른 형식의 값을 저장하는데 사용한다. 그래서 이 값에는 프로그램이 실행될 때 프로그램의 흐름에 따라 고정되는 정점의 타입의 값이 저장되게 된다.[1]

이것은 어떤 명령의 실행으로 생성될 수 있는 변수나 함수에 대한 정보를 저장하는데 타입과 크기에 대한 정보는 프로그램의 안전성을 점검하기 위한 중요한 백 데이터가 된다.

다음은 기계어 코드에서 타입을 조사하기 위한 방법이다.

⇒ 타입표현

기계어 코드에서 값을 표시하기 위한 byte, integer, word, long 등의 타입 표현방식이 있는데 이 타입은 해당 하드웨어 프로세서가 지원하는 메모리 할당에 따라 지원된다. 특히 integer 타입은 bit 단위로 표현하여 정확성을 높일 수 있다.

⇒ 서브타입(subtype) 관계

C 언어에서는 임의의 타입의 변경을 허용하는 cast 연산자를 제공한다. cast 연산자는 구조체에 대한 포인터로 많이 사용한다. 그런데 이런 cast 연산자의 사용은 타입에 대한 에러로 프로그램을 취약하게 하고 또 코드에 숨어있는 연결을 만들므로 프로그램에 대한 이해와 관리를 어렵게 만든다.

이런 구조체를 사용할 경우 구조체 필드들에 대한 식별이 필요한데 메모리 상에서 구조체

의 필드 구성을 고려하는 physical type checking[2]을 사용할 수 있다.

o 함수 호출 요약

기계어 코드에서 시스템 함수와 공유 라이브러리 함수 호출에 대한 안전성을 점검하기 위해서는 믿을 수 있는 코드의 경계선에서 함수의 상태에 대한 전후 정보에 대한 요약 정보가 필요하다. 함수 호출 전의 정보는 함수에서 사용되는 actual 파라메터의 역할을, 호출 후의 정보는 함수의 결과에 대하여 표현해야 한다.
이러한 정보는 위치정보와 타입정보, 값 비교 정보 등으로 표현할 수 있다.

o 스택 할당된 배열 정보 추출

스택에 할당된 배열 정보를 결정하는 것은 배열의 타입과 크기에 대한 정보를 알아야 하기 때문에 매우 어려운 작업이다.

typedef struct {
int f;
int g;
} s;
 	
int main() {
s a[10];
s *p = &a[0];
int i=0;
while(p<a+10) {
(p++)->f = I++;
}
}
	main:
	add %sp,-192, %sp
	add %sp, 96, %g3
	mov 0, %o0
	add %sp,176,%g2
	cmp %g3,%g3
	bgeu .LL3
	mov %g2,%o1
	.LL4:
	st %o0,[%g3]
	add %g3,8,%g3
	cmp %g3,%o1
	blu .LL4
	add %o0,1,%o0
	.LL3:
	ret1
	sub %sp,-192,%sp

표 2 : 로컬 배열의 타입과 크기 추정-1

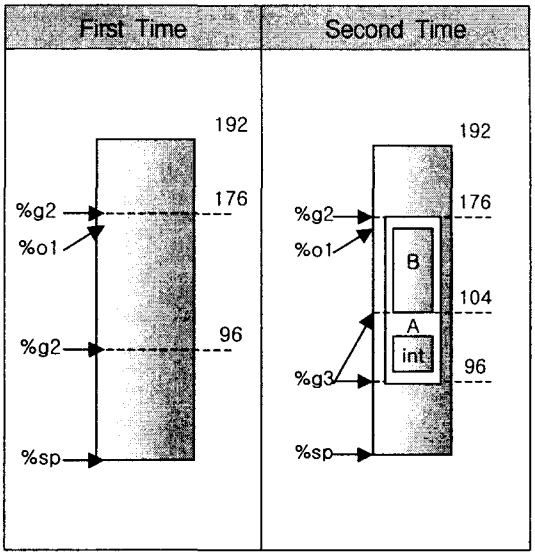


그림 2 : 로컬 배열의 타입과 크기 추정-2

먼저 스택에 할당된 배열의 범위를 알기 위해서는 루프를 시작하는 시점의 모든 포인터에 대하여 알아야 한다.

o 범위 분석

범위 분석은 프로그램의 정점에서 사용되는 각 레지스터 값의 범위를 추적하는 방법으로 배열에 대한 버퍼넘침이나 프로그램의 오류를 검증하는 방법이다.

2. 기계어 코드 수준에서의 코드의 취약점 점검 방법

언어기반 보안 점검 방법은 소스코드를 컴파일할 때 생성한 보안정보와 컴파일한 기계어 코드를 코드 검증기로 검증하는 방법이다. 이러한 방법은 소스코드와 보안정보를 가지고 검증하는데 C나 C++ 언어와 같이, 컴파일시에만 배열에 대한 정보를 참조하고 실행시에는 참조하지 않는 언어에 대

하여 보안 점검을 하는 경우에 더 정확한 탐지를 할 수 없다.

반면 기계어 코드 수준에서 코드 점검 방법은 컴파일하여 생성된 어셈블리 코드로부터 보안 정보를 추출하여 코드를 검증하는 방법이다.

이러한 코드 점검 방법은 여러 머신에 대하여 독립적인 기계어 언어를 대상으로 하므로 개발 언어에 상관없이 점검할 수 있는 장점이 있다. 또한 언어기반 보안 점검 방법에서는 점검하지 못 했던 영역까지 점검하므로 더 신뢰성이 있는 점검 방법이라고 할 수 있다.

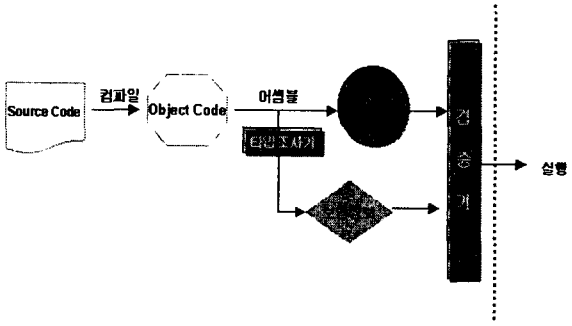


그림 2 : 기계어 코드기반 코드 검증 방법

3. 기계어 코드의 안전성 점검 구현

기계어 코드의 안전성을 점검하기 위해 기계어와 상응하는 어셈블리 언어를 이용하여 코드 안전성을 점검할 수 있도록 프로그램을 구현하였다.
위의 공개된 취약점 유형 그래프에서도 확인할 수 있듯이 버퍼오버플로우에 대한 취약점이 주를 이루므로 먼저 버퍼오버플로우 취약점만을 점검하는 프로그램으로 제한하였다. 특히 버퍼오버플로우에 취약한 시스템 콜 함수와 공유라이브러리 **함수에 전달되는 버퍼의 영역을 추적**하여 경고 메시지를 출력하게 하였다.

이러한 경고 메시지는 실제로 버퍼오버플로우의 가능성을 가지고 있는 버퍼만을 탐지하여 사용자에게 알려주므로 사용자는 버퍼의 경계값을 정확히 지정하여 버퍼오버플로우 발생 가능성을 제거할 수 있다.

1) 점검 알고리즘

기계어 코드 점검은 컴파일러의 동작처럼 2 pass 알고리즘을 사용하고 다음과 같다.

o pass 1

|. 어셈블리 코드 파일이 종료되기 전까지
  |. 명령어를 한 줄씩 읽는다.
    |. 함수가 발견되면 함수 이름과, 파일명, off_set 등을 func 에 저장한다.
    |. main 함수의 위치를 저장
  |. 다음줄을 읽는다.
|. 다음 파일로 읽는다.

o pass 2

|. main 함수 컨텍스트를 할당하여 main 함수에서 시작
  |. 명령어를 발견하면(get_opcode) 각 명령어를 파싱하여 operation code를 얻는다.
    { O_CALL, O_LEAVE, O_RET
    O_PUSH, O_POP
    O_MOV, O_ADD, O_SUB, ...}
    |. operation code의 각 operand의 모드 파악
      |. operand의 수만큼 operand를 파싱한다.
        { M_CONST : 집적(immediate, constant) 모드
        M_REG : 레지스터만 사용
        M_OFFREG1 : 1개의 레지스터를 사용한 간접주소 모드
        M_OFFREG2 : 2개의 레지스터를 사용한 간접 주소 모드
        M_LSYMOFF : 지역심볼 오프셋

값을 사용한 직접 주소모드
M_GSYMOFF : 전역심볼 오프셋 값을 사용한 직접 주소모드
M_LVAROFF : 지역변수 오프셋 값을 사용한 간접 주소모드
M_GVAROFF : 전역변수 오프셋 값을 사용한 간접 주소모드
M_LAVRREGOFF : 지역변수와 레지스터를 사용한 간접 주소모드
M_GAVRREGOFF : 전역변수와 레지스터를 사용한 간접 주소모드
}

|. operand를 파싱하여 전역변수와 지역변수에 대한 심볼테이블을 작성한다.

|. operation code에 따라 연산 수행

|. O_CALL이면 다음 함수를 컨텍스트에 할당한다.

2) 주요 자료구조

C 언어를 컴파일하여 기계어 코드를 생성하는 과정은 레지스터와 메모리간의 저장(save)과 로드(load), 그리고 레지스터간의 연산의 동작으로 이루어진다. C 언어에서 사용되는 주요 동작과정은 전역변수와 지역변수의 관리, 함수의 매개변수 전달, 서브함수 호출시 문맥(Context)의 저장과 복구를 들 수 있다.

‣ 전역변수와 지역변수

함수 밖에 선언되어 사용하는 전역변수는 서브함수에서도 접근이 가능하기 때문에 컴파일 후 변하지 않는 메모리 주소에 할당된다. 그러나 지역변수는 특정 메모리 주소에 고정되지 않고 현재의 스택 포인터의 위치에서 일정한 거리(offset)만큼 떨어진 부분의 위치에 할당되므로 서브함수에서 정의된 지역변수는 다른 함수에서는 유효하지 않다.

‣ 함수의 매개변수 전달

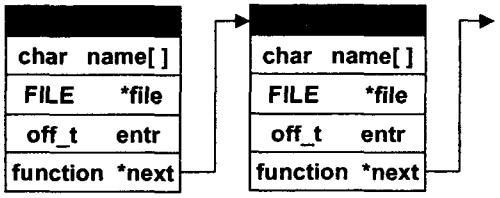
함수 호출시 매개변수를 전달하기 위하여 C 언어에서는 스택이나 레지스터에 매개변수를 저장한 뒤에 사용한다. 매개변수를 저장하기 위하여 스택을 사용하였을 경우에는 문제가 없지만 레지스터를 사용하는 경우에는 매개변수가 레지스터 수보다 많아지면 스택을 사용한다.

‣ 서브함수의 호출

함수안에서 서브함수를 호출하였을 경우 원래의 함수에서 사용하고 있던 레지스터는 서브함수에서 사용해야 하므로 레지스터의 값은 스택과 같은 지정된 장소에 저장한다. 서브함수가 실행된 후 원래의 함수로 복귀하면 스택에 저장되어 있던 값들을 다시 레지스터에 복구한다.

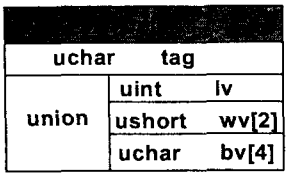
위와 같은 C 언어의 메모리 관리 구조를 만족시키기 위하여 어셈블리 언어 수준에서의 소스코드 보안취약점 점검방법에서는 다음과 같은 자료구조를 제안한다.

o function 자료구조



function 자료구조는 function이 인지될 때마다 function의 정보를 가지고 있다.

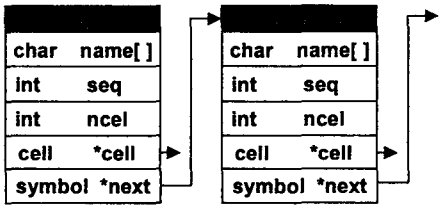
o Cell 자료구조



Cell 자료구조는 tag에 따라 메모리의 크기를 표현하기위해 사용되는 자료구조로 각 tag의 의미는 다음과 같다.

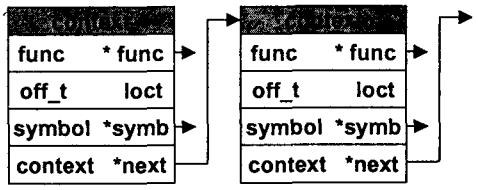
- T_VALUE : 수
- T_LPTR : 지역 심볼 값
- T_GPTR : 전역 심볼 값
- T_HPTR : malloc()과 같은 메모리 할당 함수에의하여 부여된 값
- T_RETA : call 명령이 호출되었을 경우 스택 추적을 위하여 할당
- T_STACK : 스택 영역에 설정된 변수나 매개변수등을 사용할 때 스택 자료구조의 포인터 값
- T_UNKWN : 지정되지 않은 도메인 사용시

o Symbol Table



Symbol Table은 프로그램에서 사용하는 식별자(ID)에 대한 정보를 가지고 관리하기 위한 구조체이다. 식별자로는 전역변수와 함수이름, 라벨이 될 수 있다.

o Context 자료구조



Context는 서브함수 호출시 함수의 문맥을 관리하기 위한 구조체로 operation code가 call인 경우에 할당되고 ret을 만나면 프리한다.

Ⅳ. 결론

본 논문에서는 대부분의 취약점의 원인이 되는 소스코드의 오류를 탐지하는 방법으로 어셈블리언어 레벨에서 탐지하는 방법을 제시하였다. 이 방법은 C 언어로 짜여진 프로그램을 컴파일과 어셈블 과정을 거쳐 생성된 어셈블리 코드를 입력으로 하여 버퍼오버플로우 취약점이 발생할 가능성이 있는 함수 history를 보여준다.

버퍼오버플로우 취약점은 함수호출 후 리턴할 때 잘못된 리턴주소를 지정하여 프로그램의 흐름을 바꾸는 취약점이다. 그러므로 오류 탐지시 함수 history를 추적하여 버퍼오버플로우를 허용하는 표준 라이브러리 함수가 호출되었을 때 스택에 저장되어 있는 매개변수들을 봄으로 호출된 함수에 전달되는 매개변수의 도메인을 알 수 있다.

이처럼 어셈블리 레벨에서 탐지하는 방법은 어느정도의 메모리와 스택 영역까지 탐지하므로 실제 공격에 이용되는 버퍼오버플로우 취약점을 탐지하는데 적합할 것으로 보인다.

참고문헌

[1] ZHICHEN Xu, Safety-Checking of Machine Code, University of Wisconsin-Madison, 2001

[2] Satish Chandra, THomas Reps, Physical Type Checking for C

[3] Rastislav Bodik, Rajiv Gupta, Vivek Sarkar, ABCD:Eliminating Array-Bounds Checks on Demand

[4] Dexter Kozen, Language-Based Security, Department of Computer Science Cornell University

[5] 한국정보보호진흥원, 해킹바이러스연구 00-3, 정적 소스코드 보안신뢰성에 관한 연구, 2000

[6] PARA-PROTECT, Secure Coding : The State of the Practice, 2001

[7] The Blind Leading the Blind
http://www.securityfocus.com/templates/article.html?id=247