C 언어에서 프로세서의 스택관리 형태가 프로그램 보안에 미치는 영향

이 형 봉† · 차 홍 준†† · 노 희 영†† · 이 상 민††

요 약

전통적으로 프로그램이 갖춰야 할 품질조건으로 정확성, 신뢰성, 효율성, 호환성, 이식성 등 여러 가지가 제안되어 왔지만, 최근에는 보안성이란 새로운 항목이 요구되고 있다. 보안성은 설계된 프로그램의 흐름을 사용자가 임으로 변경함으로써 보안 침해수단으로 사용하는 사례가 늘어나면서 그 중요성이 더욱 강조되고 있다. 이러한 보안 침해기법은 기본적으로 스택의 조작에서부터 출발한다. 스택과 관련된 일련의 동작들은 프로세서에 따라 고유하게 이루어지고, C 언어는 그러한 고유한 특성에 따라 스택을 관리한다. 본 논문에서는 스택 조작을 통한 보안침해의 개념을 살펴보고, 펜티엄(Pentium), 알파(Alpha), 스파크(SPARC) 등이 제공하는 스택 메커니즘을 자세히 조명해 본 후, 그 것들이 프로그램의 보안성에 어떻게 영향을 미치는지를 규명함으로써 안전 한 프로그램 작성을 위한 지침에 기여하도록 한다.

키워드 : 스택프레임, 배열경계, 버퍼넘침, 지역변수, 매개변수, 복귀주소, 펜티엄, 알파, 스파크

A Study on the Effect of Processor Stack Frame Mechanism on Secure Programming in C Language

Hyung-Bong Lee† · Hong-Jun Tcha†† · Hi-Young Rho†† · Sang-Min Rhee††

ABSTRACT

There are several traditional factors of software quality. Some of them are such as correctness, reliability, efficiency, compatibility, portability, etc. In addition to them, security is required as another factor of software quality nowadays because some application programs are used as a way to attack information systems by stack frame manipulation. Each processor has its own peculiar stack frame mechanism and C language uses the characteristics of them. This paper explains the concept of security problem caused by stack frame manipulation, and the stack frame mechanism of Pentium, Alpha and SPARC processor in detail. And then it examines the effect of stack frame mechanism on the security of programs in C language.

Key word : stack frame, boundary of array, buffer overflow, local variables, parameter, return address, Pentium, Alpha, SPARC

1. 서 론

전통적으로 소프트웨어의 품질 요소는 다양한 측면에서 제안되어 왔는데 그 중의 한가지가 McCall이 제안한 것이고, 여기서는 정확성(correctness), 신뢰성(reliability), 효율성(efficiency), 무결성(integrity), 사용성(usability), 유지보수성(maintainability), 융통성(flexibility), 시험성(estability, 이식성(portability), 재사용성(reusability), 호환성(interoperability) 등을 제안하고 있다[3]. 그러나 최근에는 소프트웨어 자체의 설계 잘못이나 구현 오류가 없더라도 언어 자체의 구조적인 특성을 사용하여 프로그램 동작 흐름을 변경시킴으로써 보안 침해를 행하는 사례가 늘어나면서 소프트웨어의 품질 요소에 보안성이란 항목이 강력히 요구되고 있다[4-7]. 위 McCall의 품질요소 중 무결성에 관한 요소가 보안성과 일맥 상통한 점이 있기는 하지만 프로그램 자체에 대한 접근통제에 주안점이 있다는 점에서 보안성과 구별된다[3].

보안성이 약한 프로그램은 사용자에 의하여 원래의 의도된 제어 범위를 벗어나 프로그램의 흐름이 변경되어 보안 침해수단으로 이용될 수 있다. 이런 유형의 보안성 침해는 사용자가 접근하거나 변경하기 쉬운 스택프레임을 조작함으로써 이루어진다[7, 8]. 스택프레임의 운영은 프로세서가 제공하는 고유한 방식을 C와 같은 고급언어가 활용하는 형태로 이루어진다. 따라서 소프트웨어의 보안 측면에서의 품질은 플랫폼 기저에 존재하는 프로세서의 스택프레임 운용 방식에 영향을 받지 않을 수 없다[2]. 본 논문의 2장에서 이러한 스택프레임의 관리 방식이 일반적인 C로 작성된 소프트웨어의 보안성과 어떤 관계를 가지는지를 관련연구를

† 종신회원 : 호남대학교 정보통신공학부 교수
†† 정 회 원 : 강원대학교 컴퓨터과학과 교수
논문접수 : 2000년 12월 7일, 심사완료 : 2001년 1월 30일

통해서 살펴보고, 3장에서는 하나의 전형적인 샘플 프로그램이 컴파일된 결과를 통하여 최근 대표적으로 많이 사용되고 있는 펜티엄(Pentium), 알파(Alpha), 스파크(SPARC) 프로세서들이 제공하는 스택프레임 메커니즘을 자세히 들여다본다. 4장에서는 이들의 고유한 스택프레임 방식의 특성 중, 2장에서 설명된 보안성에 중대한 영향을 미치는 요인을 분석하고 적용방안을 제시한다. 마지막으로 5장에서 결론으로 본 논문을 마친다.

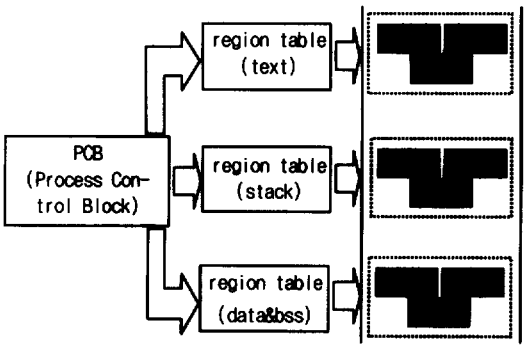
2. 관련연구

스택과 관련된 프로그램 보안성에 관한 연구는 버퍼넘침 공격(buffer overflow attack)이라는 주제로 널리 알려져 있으며[6-8], 이는 C/C++ 언어의 특징과 프로세서의 내부적인 스택관리 형태에 직접적인 영향을 받는다. 여기서는 C/C++ 언어의 일반적인 특징과 스택프레임에 대하여 살펴본다.

2.1 C 언어의 일반적인 특성

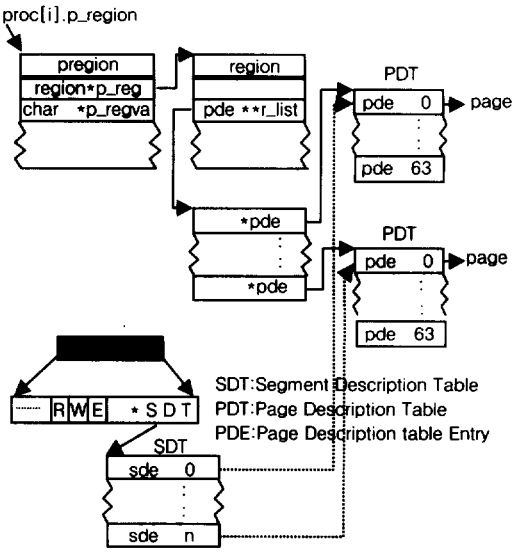
2.1.1 실행파일 이미지의 구성

실행파일의 구성은 기본적으로 실행환경을 제공하는 운영체제에 따라 다르나 대체적으로 (그림 1)과 같이 text, data &bss, stack영역으로 구성된다. 이러한 구성 형태를 포맷을 COFF(Common Object File Format), ELF(Executable or Linking Format) 혹은 a.out이라 하는데 text, data&bss, stack 각각을 리전, 섹션 혹은 세그먼트(region, section or segment)라는 단위로 관리하여 읽기·쓰기·실행 등에 관한 특성을 독립적으로 설정할 수 있다[15, 16].



(그림 1) 실행파일 이미지의 구성

이들 각 리전들은 가상 주소 공간상에서 하나의 연속된 슬롯에 대응되고 그 하위에는 (그림 2)와 같이 물리 페이지들을 연결하고 있어서 프로세서가 발산한 가상 주소가 물리 주소로 변환되는 과정에서 중간 다리 역할을 한다. 이러한 메모리 관리 구조는 프로세스 스위칭을 위한 문맥교환(context switching)에 있어서 매우 효율적인 근간으로 작용한다. 즉 실행할 프로세스의 리전 정보로부터 주소 변환에 사용되는 하드웨어 레지스터 내용을 용이하게 변경할 수 있다는 것이다[15].



(그림 2) 실행파일을 위한 메모리 관리구조

2.1.2 전역변수와 지역변수

C 언어는 변수에 대한 접근유효 규칙(scope rule)을 크게 전역변수(global variable)와 지역변수(local variable)로 이원화하여 관리한다[1]. 예를 들어 (프로그램 1)에서 함수 밖에 선언된 변수 x는 전역변수이기 때문에 함수 sub()내에서도 접근이 가능하다. 그런데 함수 main()에서는 지역변수 x가 함수 내부에 정의되어 있으므로 전역변수 x에는 접근할 수 없다.

```
int   x;
main() {
    int   x;
    char  buf[16];

    (void)sub(buf);
    x = 10;
}
sub(char arg[]) {
    x = 20;
    arg[20] = 'x';
}
```

(프로그램 1) 전역변수와 지역변수

이와 같은 전역변수와 지역변수는 단순히 접근유효 규칙상으로만 다른 것이 아니고, 내부적인 관리방식에 있어서도 큰 차이점이 존재한다. 전역변수는 컴파일 당시 메모리 위치(주소)가 항구적으로 할당되어 프로세스 이미지의 data& bss 섹션에 한 자리를 차지하기 때문에 load(save) *register,addr* 형태의 기계어로 번역된다. 그러나 지역변수는 프로그램의 진행 과정에서 설정된 스택의 일부에 위치하게

되어 load(save) *register,off*(%sp)와 같은 형태의 기계어로 번역된다. 즉 지역변수에는 특정 메모리 주소가 고정되는 것이 아니고, 현재 설정된 스택 포인터를 중심으로 일정한 거리(offset)에 있는 지점 일부영역이 할당된다. 이 때 스택 포인터는 함수가 호출·복귀를 거듭할 때마다 증·감을 반복하기 때문에 어느 함수 내에 정의된 지역변수의 위치는 일정하지 않고, 제어 흐름이 해당 함수를 떠나면 스택 포인터도 변하여 저장된 내용의 지속성도 보장할 수 없다[2]. 이런이유로 지역변수를 자동변수(automatic variable)라고도 한다.

2.1.3 함수의 매개변수 전달

C 언어는 함수를 호출할 때 필요한 매개변수 전달을 call-by-value[2] 방식으로 처리한다. 이에 따라 매개변수의 값을 일정한 장소에 저장한 뒤 함수를 호출해야 하는데 C 언어는 그 저장장소로 스택이나 지정된 레지스터를 사용한다. 그러나 매개변수의 수가 많아지면 레지스터의 수가 한정되어 있으므로 어떤 경우에도 스택 사용이 불가피하다. 즉 함수를 호출하기 전에 매개변수를 스택에 저장(push)하였다가 함수에서 복귀한 직후에 꺼내서(pop) 버리는 과정을 반복한다.

2.1.4 문맥의 저장과 복구

고급 언어인 C 원천코드가 번역되면 기계어가 생성되고, 이들의 주요 처리 내용은 프로세서 레지스터 와 메모리간의 저장(save)과 탑재(load), 그리고 레지스터들끼리의 연산 동작들이 주류를 이룬다. 이는 앞 절에서 설명한 전역 변수들은 모두 일단 레지스터를 거쳐서 계산된 후 최종 값이 부여됨을 의미한다. 이 때 이들이 최종 값에 도달하기 전의 중간 값들은 레지스터에서 잠시 머물러야 한다. 뿐만 아니라 최종 계산을 위한 임시 값(변수)를 레지스터에 보관해야 하는 경우가 많다.

위와 같은 과정에서 서브함수 호출이 일어나면, 그 곳 서브함수에서 이미 사용 중인 레지스터를 다시 사용해야 하는데, 이렇게 되면 레지스터를 안심하고 사용할 수 없게 된다. 이를 해결하기 위하여 하나의 서브함수에서 사용 중인 모든 레지스터를 지정된 장소에 보관한 후 서브 함수를 호출하고, 해당 서브함수에서 복귀하면 저장했던 레지스터들의 값을 복구하는 방법을 사용하는데 이를 문맥(context)이라고 한다[2, 15]. 여기서도 문맥의 저장장소로서 스택을 사용하는 것이 일반적이다. 특별히 다중 레지스터 집합 기법을 사용하는 경우가 있으나[17], 이 방법 역시 함수호출의 깊이가 깊어지면 저장용량에 한계를 가지므로 스택 사용이 불가피하다.

2.1.5 흐름제어 정보의 저장과 복구

C 언어 및 어셈블리 언어를 포함한 모든 언어는 서브루틴의 효율적인 호출·복귀를 위하여 서브함수 호출 시 PC (Program Counter, return address), PSW(Program Status Word), SP(Stack Pointer), FP(Frame Pointer)등 흐름제어와 관련이 깊은 정보들을 일정한 장소에 자동으로 보관하고, 복귀할 때 자동으로 복구하는 일련의 기본 동작을 필요로 한다. 이러한 기본 명령어는 프로세서의 종류에 따라 각각 고유의 기계어로 제공되기 때문에 고급 언어들은 그 범위 내에서 기능을 최대한으로 활용하여야 한다[2].

이 때, 위와 같은 제어흐름 정보의 저장 장소로서 스택을 사용하는 경우가 많고, 경우에 따라 레지스터를 사용하는 경우도 있다.

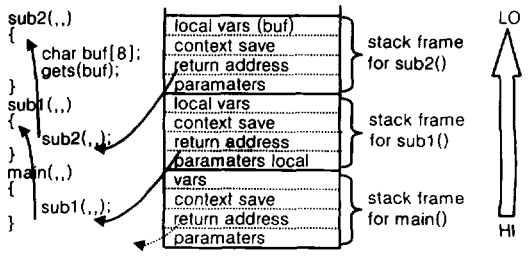
2.1.6 배열 경계의 무시

C 언어에서는 근본적으로 배열의 경계를 인식할 수 없다. 즉 배열 변수에 사용할 첨자(index)와 할당된 배열의 크기를 프로그램상에서 명시적으로 비교하지 않는 한 배열의 경계가 파괴되는 현상이 언어 수준에서 인지되지 않는다는 것이다. 이는 배열 변수를 포인터와 같은 개념으로 취급하는 데서 나타나는 현상이다. (프로그램 1)에서 sub() 함수에 전달된 배열 buf[]의 주소는 단순히 포인터로 인식되어 원래 할당된 크기인 16 바이트를 넘어선 곳을 참조하더라도 언어수준에서 어떠한 경고나 오류도 보여주지 않는다[1]. 다만 해당 프로그램을 실행할 때 논리적인 오류나 메모리 접근 오류에 의한 운영체제 수준에서의 강제적인 종료 등의 처분을 받을 수 있다[15].

이와 같이 배열의 경계가 지켜지지 않고 파괴되는 현상을 프로그램 보안 측면에서는 버퍼넘침(buffer overflow)라고 한다[6-8].

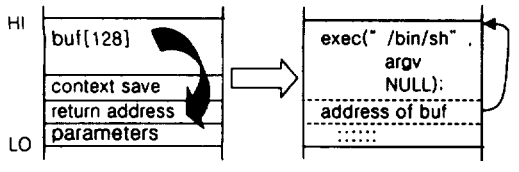
2.2 C 언어의 특성과 프로그램의 보안성

앞에서 설명한 C 언어의 여러 가지 일반적인 특성들을 스택을 중심으로 하여 종합적으로 정리하면 (그림 3)과 같이 묘사할 수 있다. 이 그림을 자세히 살펴보면 sub2() 함수에서 지역변수인 배열 buf[]가 넘칠 경우, buf 다음에 위치한 각종 프로그램 제어정보가 훼손된다는 점이다. 이와 같은 현상은 gets() 함수에서 배열 buf의 경계를 알 수 없다는 사실에 그 근본 원인이 있다. 특히 gets() 함수는 사용자의 입력을 처리하는 부분이어서 그 입력크기를 예측할 수 없기 때문에 버퍼넘침 가능성은 더욱 커지게 된다.



(그림 3) C 언어의 함수호출과 스택변화

버퍼넘침의 결과는 프로그램의 오동작이나 프로그램의 비정상적인 종료로 나타나는 경우가 대부분이다. 그러나 사용자가 이런 현상을 악용할 경우 심각한 보안상 문제에 직면하게 된다. 예를 들어 유닉스 혹은 리눅스 시스템의 경우 사용자가 터미널 입력시 (그림 4)와 같이 의도적으로 버퍼넘침을 유도할 경우 실행중이던 프로그램이 갑자기 명령어 처리기인 쉘로 돌변하게 된다. 즉, 복귀주소(return address) 부분에 배열 buf의 주소가 들어가게 하면 제어가 일단 buf 부분으로 흘러 그곳에 위치한 명령어(기계어)들을 수행하게 될 것인데, 그 곳(buf 부분)에 사용자가 원하는 기계어 코드(쉘 생성 코드)가 위치하도록 입력문자열을 배치하면 버퍼넘침을 사용한 보안침해가 이루어지게 된다. 이 때, 원래의 프로그램에 관리자의 권한위임(setuid, setgid)이 설정되어 있었다면 그 결과는 매우 심각하다[8, 9].



(그림 4) 의도적인 버퍼넘침의 유도

이와 같은 버퍼넘침에 대한 연구는 주로 전통적인 스택 관리 환경 즉, 서브함수 호출시 매개변수, 제어정보, 문맥 등이 모두 스택에서 관리된다는 전제 하에 순전히 소프트웨어(운영체제, 라이브러리, 컴파일러 등) 수준에서 이루어지고 있다. [8, 9]는 펜티엄 기반의 리눅스 시스템에서 GNU C를 사용하여 위와 같은 보안침해가 이루어지는 과정을 실증적으로 다루고 있고, [10]은 리눅스 운영체제의 스택 세그먼트에서 실행 모드를 제거하는 방안을, [9, 11, 12]는 버퍼넘침을 불허하거나 복귀시 버퍼넘침을 탐지할 수 있는 라이브러리 수준의 예방 방안을, 그리고 [13, 14]는 컴파일시 정적으로 버퍼넘침 가능성을 분석하여 탐지하는 방안을 각각 제시하고 있다. 이와 같은 연구 경향은 특정 벤더에 대한 종속 지양, 개방화 추구, 보편적이고 독립적인 해결방안을 추구하려는 의지에서 비롯된 것으로 보이지만, 완전한 해결책이 되지 못하거나 성능 등 다른 측면에서의 부작용을 동반하고 있다[9]. 이런 관점에서, 한 차원 낮은 하드웨어(프로세서)의 특성이 프로그램의 보안성에 미치는 영향을 규명하고, 그러한 영향을 활용한 해결책을 모색해 보는 것은 매우 의미있는 일이 아닐 수 없을 것이다.

3. 프로세서의 스택관리 형태에 대한 고찰

(그림 3), (그림 4)에서 제시한 스택프레임 형태는 전통적이고 일반적인 프로세서를 가정하고 있다. 그러나 최근 RISC(Reduced Instruction Set Computer) 기술의 발전에 따라 메모리 접근을 가능한 한 배제하려는 추세가 뚜렷해지고 있는데[17], 이는 메모리 접근 속도가 상대적으로 매우 느리기 때문이다. 이에 따라 함수 호출을 위한 스택관리 정책에 있어서도 제어 정보(복귀주소나 매개변수 등)을 스택이 아닌 레지스터에 저장하는 프로세서가 급증하고 있다.

위와 같이 스택 프래임의 관리 정책이 변할 경우 프로그램 보안에 미치는 버퍼넘침의 영향이 크게 달라질 수 있다. 예를 들어 복귀주소가 스택에 존재하지 않는다면 의도적인 제어흐름의 변경이 거의 불가능해 진다. 이는 프로세서의 스택관리의 형태에 따라 프로그램의 원천적인 보안성이 크게 좌우될 수 있음을 의미한다.

여기서는 최근 많이 사용되는 펜티엄(Pentium)[18], 알파(Alpha)[19], 스파크(SPARC)[20]를 대표적인 프로세서로 선정하여, 유닉스 혹은 리눅스 시스템상의 간단한 C 프로그램을 통하여 그들의 스택관리 방식을 자세히 조명함으로써, 그들이 가지고 있는 프로그램 보안성에 대한 영향 인자들을 도출한다.

3.1 스택관리 방식 분석을 위한 C 프로그램

(프로그램 2)는 위의 각 프로세서의 스택관리 형태를 관찰하기 위해 작성한 전형적인 샘플 프로그램으로, 사용자가 단말기로부터 입력한 두 개의 숫자 열을 이진수(binary number)로 변환하여 더한 후 그 결과를 출력하는 내용이다. 이 프로그램 내용은 버퍼 넘침을 유발할 가능성이 있고, 함수 호출 과정을 살펴보기에 알맞도록 작성되었다.

```
 1  #include <stdio.h>
 2
 3  int main(int ac, char *av[]) {
 4      int         sum;
 5
 6      if (ac != 3)
 7          return;
 8
 9      sum = add(av[1], av[2]);
10      printf("%s+%s = %d\n", av[1], av[2], sum);
11
12      return;
13  }
14
15  int add(char *bf1, char *bf2) {
16      char        buf1[8], buf2[8];
17      int         num1, num2, sum;
18
19      strcpy(buf1, bf1);
20      strcpy(buf2, bf2);
21
22      num1 = atoi(bf1);
23      num2 = atoi(bf2);
24
25      sum  = num1 + num2;
26      return(sum);
27  }
```

(프로그램 2) 스택 관리방식 관찰을 위한 프로그램

대부분의 C 컴파일러는 어셈블리 코드를 생성하는 옵션을 가지고 있고(<표 1> 참조), 어셈블리 코드를 분석하면 해당 C 컴파일러 및 프로세서의 스택관리 형태에 대한 정보를 역으로 얻어낼 수 있다. 본 논문에서도 각 프로세서가 장착된 시스템에서, (프로그램 2)에 대응되는 어셈블리 코드를 생성하여 각 컴파일러 및 프로세서의 스택관리 형태를 프로그램 줄 번호와 그림을 대응시켜가며 세밀하게 분석하였다.

〈표 1〉 (프로그램 2)의 분석환경

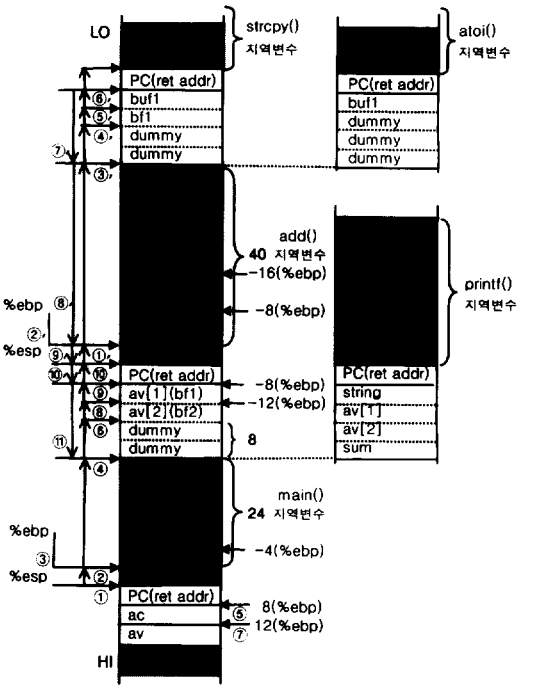
프로세서	시스템	운영체제	컴파일러	옵션
펜티엄(Pentium)	PC	LINUX	GNU cc	-S
알파(Alpha)	COMPAQ WS	T r u 6 4 UNIX	COMPAQ cc	-S
스파크(SPARC)	SUN WS	Solaris	SUN cc	-S

3.2 펜티엄 프로세서의 스택관리 방식

(프로그램 3)은 <표 1>에 따라 (프로그램 2)를 리눅스 시스템에서 GNU C 컴파일러를 사용해 생성한 펜티엄 어셈블리 코드를, (그림 5)는 이 프로그램의 진행에 따른 스택의 변화과정을 각각 보여주고 있다.

```
 :
 5  .LC0:
 6      .string  "%s+%s = %d\n"
 :
11  main:
12      pushl   %ebp
13      movl    %esp,%ebp
 :
15      cmpl    $3,8(%ebp)
16      je      .L3
 :
19  .L3:
20      addl    $-8,%esp
21      movl    12(%ebp),%eax
22      addl    $8,%eax
23      movl    (%eax),%edx
24      pushl   %edx
25      movl    12(%ebp),%eax
26      addl    $4,%eax
27      movl    (%eax),%edx
28      pushl   %edx
29      call    add
30      addl    $16,%esp
31      movl    %eax,-4(%ebp)
 :
49      leave
50      ret
 :
56  add:
57      pushl   %ebp
58      movl    %esp,%ebp
59      subl    $40,%esp
60      addl    $-8,%esp
61      movl    8(%ebp),%eax
62      pushl   %eax
63      leal    -8(%ebp),%eax
64      pushl   %eax
65      call    strcpy
66      addl    $16,%esp
67      addl    $-8,%esp
68      movl    12(%ebp),%eax
69      pushl   %eax
70      leal    -16(%ebp),%eax
71      pushl   %eax
72      call    strcpy
73      addl    $16,%esp
74      addl    $-12,%esp
75      movl    8(%ebp),%eax
76      pushl   %eax
77      call    atoi
78      addl    $16,%esp
79      movl    %eax,%eax
80      movl    %eax,-20(%ebp)
81      addl    $-12,%esp
82      movl    12(%ebp),%eax
83      pushl   %eax
84      call    atoi
85      addl    $16,%esp
86      movl    %eax,%eax
87      movl    %eax,-24(%ebp)
88      movl    -20(%ebp),%eax
89      movl    -24(%ebp),%edx
90      leal    (%edx,%eax),%ecx
91      movl    %ecx,-28(%ebp)
92      movl    -28(%ebp),%edx
93      movl    %edx,%eax
 :
97      leave
98      ret
 :
```

(프로그램 3) (프로그램 2)의 펜티엄 어셈블리 코드



(그림 5) (프로그램 3)의 진행에 따른 스택 변화과정

● 스택프레임의 유지

펜티엄 프로세서는 스택포인터 %esp와 프레임포인터 %ebp를 사용하여 스택프레임을 유지한다. (프로그램 3)의 12-13, 57-58 번 줄은 호출한 함수에서 사용 중이던 스택프레임 %ebp를 저장하고, 현재 호출된 함수의 스택프레임을 새롭게 설정하는 과정을 보여주고 있다(① → ②, ①' → ②'). 이와 반대로 호출된 함수에서 복귀할 때는 49, 97번 줄에서와 같이 leave 명령어(instruction)에 의해 이전 함수의 스택프레임을 복구시킨다. leave 명령어는 %ebp 값을 %esp에 지정한 후 스택에서 pop한 값을 다시 %ebp에 지정하는 연산을 연속적으로 수행한다.

● 지역변수 영역의 확보 및 접근

14, 59번 줄에서는 스택포인터를 확장 시킴으로써 스택에 할당될 지역변수 공간을 할당하고 있는데, 할당영역의 크기는 해당 함수에서 정의된 지역변수 들의 크기와 수에 따라 다르다(② → ④, ①' → ③').

31번 줄은 add()의 함수 값을 지역변수 sum에 지정하는 문장인데, sum 변수가 위에서 할당한 지역변수 영역 내에 위치함을 알 수 있다. 63, 70, 80, 87, 87, 91번 줄은 각각 add() 함수 내의 buf1, buf2, num1, num2, sum 변수들을 사용하는 과정을 보여주고 있다.

● 매개변수의 전달

15번 줄에서는 %ebp부터 아래로8만큼 떨어진 위치(⑤)의 값을 사용하고 있는데, 그 것이 바로 main() 함수에 넘겨진 매개변수 ac에 대응된다. 이는 매개변수가 스택을 통해서

전달됨을 의미한다. 21, 25번 줄에서는 같은 방법으로 매개변수 av의 값을 %ebp에서 12 만큼 떨어진 곳(⑦)에서 얻고 있는데, 이는 매개변수의 순서가 스택에 역순으로 전달됨을 보여준다.

24, 28번 줄에서는 add() 함수를 호출하기 위해 매개변수 av[2], av[1]을 차례로 스택(⑥→⑧, ⑧→⑨)에 저장하고 있다. 여기서 유의할 사항은 20번 줄에서 8 바이트의 스택 공간을 버리고 있다는 점인데(④→⑥), 이는 호출된 다음 함수의 스택프레임이 16의 배수가 될 수 있도록 배려하기 위함이다.

이와 같이 매개변수를 위해 사용된 스택영역은 호출된 함수에서 복귀한 직후 반납해야 한다. 줄 30은 서브함수 add()의 매개변수로 사용되었던 스택영역을 반납하고 있다(⑩→⑪).

● 복귀주소의 관리

29번 줄은 서브함수 add()를 호출하고 있는데, 이 때 call 명령어는 현재의 PC 값을 스택에 보관(⑨→⑩)하고 해당 위치로 분기한다. 여기서 보관된 PC 값은 98번 줄의 ret 명령어에 의해 복구되어(⑨'→⑩') 원래 함수로 되돌아 온다.

3.3 알파 프로세서의 스택관리 방식

(프로그램 4)는 <표 1>에 따라 (프로그램 2)를 유닉스 시스템에서 COMPAQ C 컴파일러를 사용해 생성한 알파 어셈블리 코드를, (그림 6)은 이 프로그램의 진행에 따른 스택의 변화과정을 보여주고 있다. 알파 프로세서에는 모두 31개의 범용 레지스터가 있고 COMPAQ C는 $16-21까지를 함수 호출시 첫 6개의 매개변수 전달용으로, $26 레지스터는 복귀주소를 저장하는 용도로 사용하는데, 이들 레지스터 값들은 호출된 함수에서 또 다른 함수를 호출할 경우에는 스택에 보관되어야 한다. 함수에서 복귀할 때 단순 복귀값은 $0에 반환된다. 그 밖에 $1-8, $22-25레지스터들은 임시 처리용으로 각 함수에서 자유롭게 사용하고, $9- 14까지의 레지스터들은 함수에서 사용 후 복귀할 때 반드시 원상복귀(스택을 사용함)하여 사용한다. 또한 알파 프로세서는 64bit 이기 때문에 레지스터 및 스택을 포함한 메모리 접근이 8 바이트 단위로 이루어진다[19].

● 스택프레임의 유지

알파에서는 별도의 프레임포인터를 사용하지 않고 스택포인터(%sp) 만으로 프레임을 유지하기 때문에 진행 중 스택에 대한 push·pop을 사용하지 않고, 미리 확보된 스택 영역에 load·store를 사용한다.

● 지역변수 영역의 확보 및 접근

지역변수 영역의 확보 및 접근 방법은 펜티엄의 경우에서와 동일하다. 줄 28에서 main() 함수의 지역변수 영역을 위하여 스택을 확장하였다가(①→②), 64번 줄에서 다시 축소

```
 :
 5 $$2 :
 6     .ascii "%s+%s=.%d\X00"
 7
 :
19 main:
 :
28     lda   $sp,-16($sp)
29     stq   $26,($sp)
 :
31     xor   $16,3,$16
 :
33     stq   $9, 8($sp)
 :
38     mov   $17,$9
 :
40     beq   $16,L$6
41     br    L$8
 :
43 L$6:
 :
45     ldq   $16,8($9)
46     ldq   $17,16($9)
47     bsr   $26,L$2
 :
49     ldq   $27,printf
50     ldq   $17,8($9)
51     ldq   $18,16($9)
52     ldq   $16,$$2($gp)
 :
54     mov   $0,$19
 :
56     jsr   $26,($27)
 :
60 L$8:
61     ldq   $26,($sp)
62     ldq   $9,8($sp)
63     clr   $0
64     lda   $sp,16($sp)
65     ret   ($26)
 :
71     add:
 :
76 L$2:
 :
78     ldq   $27,strcpy
 :
80     lda   $sp,-48($sp)
81     stq   $16,24($sp)
 :
83     lda   $16,16($sp)
 :
85     stq   $26,($sp)
 :
90     stq   $17,32($sp)
 :
92     ldq   $17,24($sp)
93     jsr   $26,($27)
 :
95     ldq   $17,32($sp)
 :
100    lda   $16,8($sp)
101    ldq   $27,strcpy
102    jsr   $26,($27)
 :
104    ldq   $16,24($sp)
 :
109    ldq   $27,atoi
110    jsr   $26,($27)
 :
112    ldq   $16,32($sp)
 :
115    stl   $0,24($sp)
116    lda   $gp,4($gp)
 :
118    ldq   $27,atoi
119    jsr   $26,($27)
 :
121    ldl   $1,24($sp)
 :
125    ldq   $26,($sp)
 :
129    addl  $1,$0,$0
 :
131    lda   $sp,48($sp)
132    ret   ($26)
 :
```

(프로그램 4) (프로그램 2)의 알파 어셈블리 코드

하는 모습(④→⑤)을 볼 수 있다. add() 함수에서도 80번 줄에서 확장한 후(②→③), 복귀하기 직전인 줄 131에서 원위치 시키고 있다(③→④).

main() 함수내의 지역변수 sum은 add()의 복귀 값을 가지고 있는 $0를 재사용하여 스택에 할당하지 않았고, 83, 100번 줄은 add() 함수내의 지역변수 buf1, buf2에 접근하는 연산을 보여주고 있다.

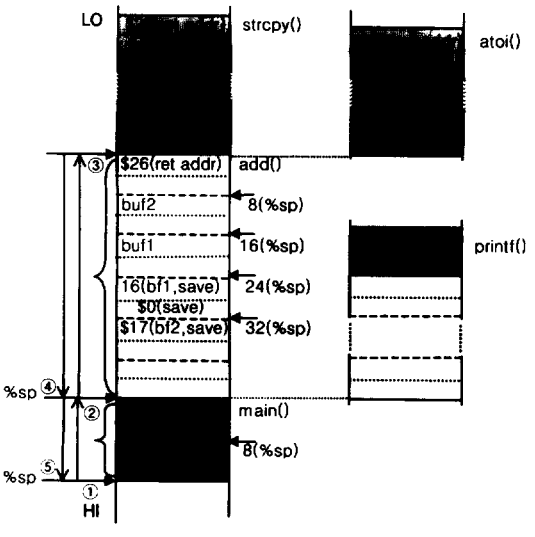
● 문맥의 저장과 복구

main() 함수는 $9를 사용하기 위하여 33번 줄에서 $9의 현재 값을 스택에 임시 저장하였다가, 복귀하기 직전인 62번 줄에서 다시 복구한다. 81, 90번 줄은 함수 add()가 $16,

$17을 다음 함수 호출에 사용하기 위하여 임시 저장하고, 104, 95번 줄은 그 값들을 다시 사용하는 과정을 보이고 있다.

● 매개변수의 전달

줄 31, 40은 main() 함수가 자신에게 전달된 첫번째 매개변수 ac를 $16으로 접근함을 보이고 있다. 38번 줄에서는 두 번째 매개변수 av를 $9에 임시 보관하였다가, 45-46번 줄에서 add() 함수 호출을 위한 매개변수를 av[1], av[2]를 각각 $16, $17에 설정하는 과정을 보이고 있다. add() 함수 내에서도 strcpy(), atoi() 함수를 위한 매개변수를 각각 같은 방법으로 설정함을 알 수 있다(줄83, 92, 95,100, 104, 112). 이 때 $16, $17등은 중복 사용됨으로 필요시 문맥으로 저장되어야 한다.



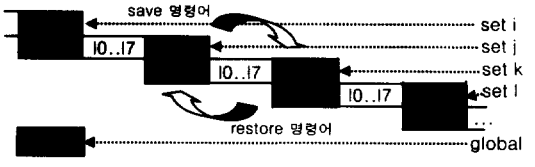
(그림 6) (프로그램 4)의 진행에 따른 스택 변화과정

● 복귀주소의 관리

줄 47, 56, 93, 102, 110, 119는 각각 서브함수를 호출하는 명령어를 포함하고 있는데, 이들은 모두 복귀주소를 스택이 아닌 $26에 저장한다. 그러나 말단 이 아닌 함수에서는 또 다른 함수를 호출하기 전에 $26값은 충돌을 방지하기 위하여 스택에 저장한다. 29번 줄은 main() 함수가 add()나 printf()를 호출할 때 $26을 사용하기 위하여 그 값을 스택에 저장하는 명령어이고, 줄 61은 복귀하기 직전에 $26 값을 복구하는 명령어이다. 줄 65, 132는 함수에서 복귀하는 ret 명령어로서 $26에서 복귀주소를 얻고 있다.

3.4 스파크 프로세서의 스택관리 방식

스파크 프로세서의 가장 큰 특징은 다중 레지스터 집합을 사용한다는 데 있다. 다중 레지스터 집합은 함수 호출시 레지스터의 보관 필요성을 줄이고 매개변수를 효과적으로 전달하기 위해 고안된 것인데, (그림 7)과 같이 전역(global), 지역(local), 중첩(overlap) 레지스터로 구성된다. 전역 레지스터는 함수호출과 관계없이 사용할 수 있고 g0~g7로 명명하고 있다. 지역 레지스터는 함수호출 때마다 수행하는 save 명령어에 따라 l0~l7까지의 레지스터군을 7개의 서로 다른 윈도우를 통해서 독립적으로 사용할 수 있다. 중첩 레지스터는 지역 레지스터와 같으나 서로 연속 되는 윈도우 사이에서 중첩된다는 점에서 차이가 있다. 즉 o0~o7 레지스터들은 save 명령어 후 다음 레지스터 윈도우에서 i0~i7까지의 레지스터와 정확히 대응된다. 따라서 o0~o7 레지스터에 매개변수를 넣고, 호출된 함수에서 save 명령어 다음에 i0~i7 레지스터를 사용하면 매개변수를 얻을 수 있고, 함수에서 복귀할 때는 반대 방향으로 값을 전달받을 수 있다. 대부분의 경우 i0 레지스터에 함수 값을 넣고 복귀 후 o0에서 반환 값을 사용한다. 이와 같은 특성을 이용하여 복귀주소를 스택 대신에 o7 레지스터에 저장하고, 복귀할 때 i7 레지스터를 사용한다[20].



(그림 7) 스파크 프로세서의 다중 레지스터 개념

(프로그램 5)는 <표 1>에 따라 (프로그램 2)를 유닉스 시스템에서 SUN C 컴파일러를 사용해 생성한 스파크 어셈블리 코드를, (그림 8)은 이 프로그램이 진행하면서 조작하는 스택의 모습을 나타내고 있다.

● 스택프레임의 유지

스파크의 스택프레임은 %fp(=%i6), %sp(=%o6)에 의하여 유지된다. 8번 줄의 save는 복합 명령어로서 윈도우가 바뀌기 전 %sp 값에 -104를 더한 값을 윈도우가 바뀐 후의 %sp(새로운 %o6, ②)에 설정한다. 따라서 save 이후에 %fp는 save이전의 %sp 값(①)을 유지하는 결과가 된다. 이와 반대로 57번 줄의 restore 명령어는 윈도우를 단순히 save 이전으로 되돌려 주는데(%sp : ②→⑦, %fp : ①→⑧), 이는 곧 현재 호출된 윈도우의 스택프레임이 %fp위에 형성되며, %fp 값은 변경되지 않음을 의미한다. add() 함수에서도 이와 동일한 과정이 반복된다(%sp : ③→⑤, %fp : ④→⑥).

● 지역변수 영역의 확보 및 접근

지역변수 및 기타 임시저장 영역의 확보 및 지역변수에 접근하는 방법은 펜티엄, 알파에서와 마찬가지로 스택프레임을 이용하여 이루어진다. 8, 68번 줄에서는 save 명령어를 사용하여 각각의 함수에서 필요한 스택을 확장하고, 줄 37, 47, 92, 99, 104, 107, 112 등에서는 %fp를 기준으로 할당된 지역변수에 접근하고 있으며, 57, 115번 줄의 restore 명령어로 스택프레임을 원위치시키고 있다.

```
 :
 7      main :
 8      save    %sp,-104,%sp
 9      st      %i1,[%fp+72]
10      st      %i0,[%fp+68]
 :
13  .L59 :
14      ld      [%fp+68],%l0
15      cmp     %l0,3
16      be      .L60
 :
25  .L60 :
26      ld      [%fp+72],%l0
27      add     %l0,4,%l0
28      ld      [%l0+0],%l1
29      ld      [%fp+72],%l0
30      add     %l0,8,%l0
31      ld      [%l0+0],%l0
32      mov     %l1,%o0
33      mov     %l0,%o1
34      call    add
35      nop
36      mov     %o0,%l0
37      st      %l0,[%fp-8]
 :
41      ld      [%fp+72],%l0
42      add     %l0,4,%l0
43      ld      [%l0+0],%l2
44      ld      [%fp+72],%l0
45      add     %l0,8,%l0
46      ld      [%l0+0],%l0 :
47      ld      [%fp-8],%l3
48      mov     %l1,%o0
49      mov     %l2,%o1
50      mov     %l0,%o2
51      mov     %l3,%o3
52      call    printf
53      nop
 :
56      jmp     %i7+8
57      restore
 :
67  add:
68      save    %sp,-128,%sp
69      st      %i1,[%fp+72]
70      st      %i0,[%fp+68]
71
72  .L66:
73      add     %fp,-12,%l0
74      ld      [%fp+68],%l1
75      mov     %l0,%o0
76      mov     %l1,%o1
77      call    strcpy
78      nop
79
80      add     %fp,-20,%l0
81      ld      [%fp+72],%l1
82      mov     %l0,%o0
83      mov     %l1,%o1
84      call    strcpy
85      nop
86
87      ld      [%fp+68],%l0
88      mov     %l0,%o0
89      call    atoi
90      nop
91      mov     %o0,%l0
92      st      %l0,[%fp-24]
93
94      ld      [%fp+72],%l0
95      mov     %l0,%o0
96      call    atoi
97      nop
98      mov     %o0,%l0
99      st      %l0,[%fp-28]
100
101     ld      [%fp-24],%l0
102     ld      [%fp-28],%l1
103     add     %l0,%l1,%l0
104     st      %l0,[%fp-32]
105
106     ld      [%fp-32],%l0
107     st      %l0,[%fp-4]
 :
112     ld      [%fp-4],%l0
113     mov     %l0,%i0
114     jmp     %i7+8
115     restore
```

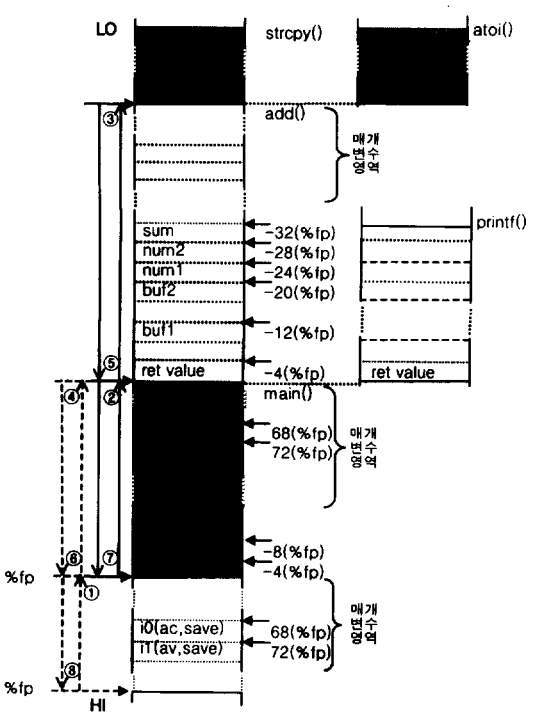
(프로그램 5) (프로그램 2)의 스파크 어셈블리 코드

● 매개변수의 전달

줄 9, 10은 main() 함수에 전달된 두개의 매개변수 ac, av를 각각 사용하는 모습이고, 32, 33번 줄은 add() 함수를 호출하기 위한 매개변수를 %o0, %o1에 각각 설정하는 문장을 보여주고 있는데 이는 매개변수 전달시 스택을 사용하지 않는다는 것을 의미한다.

● 복귀주소의 관리

스파크의 복귀주소 관리는 알파의 경우와 마찬가지로 레지스터를 사용한다. 34, 52, 77, 89번 줄 등의 call 명령어는 복귀주소를 %o7에 보관한 후 분기하고, 56, 114번 줄에서는 %i7을 이용하여 복귀하는 과정을 보이고 있다. 여기서 독특한 점은 줄 56, 114의 복귀 명령어 이후에 restore가 있다는 점인데, 이는 파이프라이닝을 효과적으로 지원하기 위한 스파크의 특성 때문이다. 즉 스파크의 모든 분기 명령어



(그림 8) (프로그램 5)의 진행에 따른 스택 변화과정

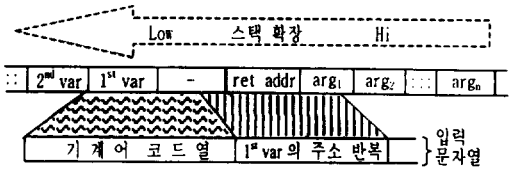
는 바로 다음 명령어까지 수행한 다음에 실제 분기가 이루어진다는 것이다. 따라서 복귀주소는 분기할 때 바로 다음 명령어를 이미 수행했으므로 그 다음 명령어의 위치인 %i7+8이 되어야 한다.

4. 프로세서의 스택관리 형태와 프로그램의 보안성

프로그램 보안상 버퍼넘침이 가지는 가장 큰 위험성은 주변의 제어정보에 해를 끼침으로써 사용자의 의도적인 흐름변경이 가능하다는 점을 2장에서 설명하였다. 3장에서는 상용화된 대표적인 프로세서들의 스택관리 방식을 자세히 살펴보았다. 여기서는 위와 같은 버퍼넘침의 위험성에 프로세서의 스택관리 방식이 미치는 영향을 분석한다.

4.1 버퍼넘침 보안침해를 시도하기 위한 절차

버퍼넘침을 유도하기 위해서는 (그림 9)와 같이 스택에 위치한 버퍼영역 앞 부분에 기계어 코드를 배치하고, 복귀주소가 저장된 부분을 버퍼의 주소 값이 덮어쓸 수 있도



(그림 9) 버퍼넘침을 유도하는 입력문자열의 구성

록 해야한다. 여기서 기계어 코드는 침해자가 원하는 위해 프로그램을 기계어로 표현한 것으로, 어셈블러나 디버거 등을 이용하여 쉽게 구할 수 있다. gets() 함수 등에서 이와 같은 입력 문자열로 버퍼가 넘치면 해당함수에서 복귀할 때 버퍼의 주소로 분기가 일어나 버퍼에 존재하는 기계어를 수행하게 된다.

4.2 버퍼넘침 보안침해를 위한 전제조건

위의 버퍼넘침 보안침해가 성립하기 위해서는 다음과 같은 전제 조건들이 필요하다.

● 복귀주소가 스택에 저장되어 있어야 함

복귀주소가 스택에 보관되지 않는다면, 버퍼넘침을 통해서 복귀주소를 인위적으로 변경시킬 수 없고, 따라서 인위적인 제어흐름의 변경이 불가능하다.

● 버퍼가 스택영역에 위치해야 함

버퍼가 스택에 위치하지 않으면 버퍼가 넘치더라도 스택에 위치한 제어정보는 어떤 영향도 받지 않는다.

● 버퍼의 위치를 알아야 함

버퍼에 입력한 기계어코드를 실행시키기 위해서는 복귀주소가 버퍼의 주소 값으로 조작되어야 하기 때문에 버퍼 자체의 주소를 알아야 한다. 뿐만 아니라 기계어코드 부분에 절대주소가 사용되는 부분이 있다면, 시작위치를 고려하여 코드를 다시 작성하여야 한다. 그런데 스택에 위치한 버퍼의 주소는 (그림 9)와 같은 입력문자열에 다양한 주소 값을 시행착오로 적용함으로써 알아낼 수 있다.

● 버퍼의 크기를 알아야 함

(그림 9)의 입력 문자열에서 버퍼의 주소 값이 복귀주소 부분을 정확히 덮어쓸 수 있기 위해서는 버퍼의 크기를 알아야 한다. 버퍼의 크기도 버퍼의 위치와 마찬가지로 시행착오를 거쳐 알아낼 수 있다.

● 스택 세그먼트에 대한 실행모드가 있어야 함

스택영역에 위치한 버퍼에 들어있는 기계어 코드가 실행되기 위해서는 스택 세그먼트에 실행모드가 설정되어 있어야 하는데, 이는 시스템 공급자에 따라 상이하고 리눅스의 경우 대부분 설정되어 있다.

4.3 프로세서의 스택관리 형태가 버퍼넘침에 미치는 영향

3장에서 살펴본 프로세서의 스택관리 형태가 앞에서 제시한 버퍼넘침 보안침해 요건에 미치는 영향은 다음과 같이 규명된다.

● 복귀주소의 저장방식에 따른 영향 및 대응

세 프로세서의 스택관리 방식에서 나타나는 가장 뚜렷한 차이점 중의 하나는 복귀주소를 어떻게 관리하는가 하는 점이다. 펜티엄은 전통적이고 규칙적인 절차에 따라 스택에 보관하고, 알파는 레지스터에 보관하고 있다가 다음 단계 함수 호출이 더 필요할 경우 스택에 보관하며, 스파크는 스택을 전혀 사용하지 않는다. 이는 앞 절에서 제시한 첫 번째 요건과 관련이 있는 사항으로 스파크의 경우 [7,8]에 의한 버퍼넘침 보안침해가 불가능함을 의미한다. 그러나 함수 호출 단계가 깊어지면 스택사용이 불가피하므로 스파크 환경에서의 C 프로그램 작성자는 버퍼넘침의 가능성이 있는 함수의 호출단계(깊이)를 정적으로 점검함으로써 버퍼넘침 보안침해를 효율적으로 예방할 수 있다.

● 복귀주소의 저장위치에 따른 영향 및 대응

복귀주소가 스택에 존재하더라도 알파의 스택관리 방식에서와 같이 그 위치가 지역 변수들의 위치보다 낮은 곳에 존재한다면 스택조작을 통한 보안 침해가 좀더 어렵게 된다. (그림 6)의 add() 함수에서 지역변수 buf1이 넘쳐 훼손되는 복귀주소는 함수호출 과정상 한단계 이전의 복귀주소 즉 main() 함수에서 복귀할 주소를 해치게 됨을 알 수 있다. 이렇게 되면 바로 이전 함수(main())로의 복귀는 정상적으로 이루어질 것이다. 그러나 복귀한 main() 함수에서 다시 다른 함수를 호출한다면 사용자의 위해코드가 들어있는 스택부분이 다른 함수의 스택프레임으로 재사용되어 그 부분의 내용이 변경될 것이기 때문에 침해자의 의도가 보존될 가능성이 크게 줄어든다. 즉 버퍼넘침의 가능성이 있는 함수의 호출 다음에는 반드시 또 다른 함수 호출을 배치한다는 것이다. 따라서 알파 환경에서의 C 프로그램 작성자는 이와 같은 함수 호출순서를 정적으로 점검함으로써 앞 절에서와 같이 효과적인 버퍼넘침 보안침해를 예방할 수 있을 것이다.

● 매개변수의 전달 방식에 따른 영향 및 대응

매개변수의 전달 방식에는 레지스터를 이용하는 방법과 스택을 이용하는 방법이 있다. 펜티엄은 전형적으로 스택을 이용하고, 알파와 스파크는 레지스터를 사용하고 있다. 이는 근본적으로RISC 프로세서의 레지스터 수가 많다는 데서 기인한 것인데, 스파크에서는 전달 받은 매개변수를 스택에 다시 저장하여 사용하고, 알파의 경우 필요한 경우만 스택에 저장하여 사용하고 있음을 알 수 있다. 매개변수를 정형적으로 스택에 저장하는 방식보다는 경우에 따라 필요한 경우에만 스택을 사용하는 방식이 프로그램 보안성에 유리할 것이다. 이는 앞 절에서 제시한 버퍼의 위치와 크기를 예측하는데 있어서 불규칙성을 높일 뿐 아니라, 스택공간을 가능한 적게 사용함으로써 악성코드의 삽입에 있어서 제한 점으로 작용할 수 있기 때문이다. 따라서 알파나 스파크 환경에서 매개변수의 전달의 불규칙성을 높이는 한가지 방안은 버퍼넘침의 가능성이 있는 함수의 매개변수 수를 가능한 한 6~7개 이하로 제한하는 것인데, 이는 첫 6~7

개의 매개변수만이 레지스터를 통해서 전달되기 때문이다. 이와 같은 권고사항도 정적 도구에 의하여 검증을 받음으로써 버퍼넘침 보안침해 예방에 기여할 수 있다.

● 파이프라인을 위한 분기의 특이성에 따른 영향

RISC 프로세서의 경우 명령어의 파이프라인 처리를 최대한으로 높이는데 있어서 장애 요소중의 하나인 분기 명령어 사용시 일반CISC 프로세서와는 다른 형태의 특별한 배려를 프로그래머에게 요구한다. 스파크의 경우 분기 명령어에서 분기가 즉시 일어나지 않고, 그 다음에 위치한 명령어 하나를 더 처리한 후 분기기 일어남을 알 수 있다. 거기에다 호출 분기의 경우 복귀주소에는 그러한 사항이 고려되지 않기 때문에, 복귀할 때는 프로그래머가 그 사실을 고려하여 복귀 주소를 재설정해야 한다.

위와 같은 분기 명령어의 특이성은 일반 프로그래밍에는 복잡하고 부담이 되지만, 기본적으로 분기에 근거하는 프로그램 보안성 측면에서는 오히려 장점으로 작용할 수 있다.

위와 같은 몇 가지 관점을 기준으로, 3장에서 살펴본 여러 프로세서들의 스택관리 형태의 특성들이 버퍼넘침 보안침해에 작용하는 긍정적·부정적 영향을 요약하면 <표 2>와 같다.

〈표 2〉 프로그램 보안성에 미치는 프로세서 특징

특성항목	펜티엄	알파	스파크
복귀주소 관리방식	반드시 스택에 저장	레지스터에 저장 후 스택에 저장	반드시 레지스터에 저장
	단 점	장점	단 점
복귀주소 위 치	지역변수와 근거리	지역변수와 원거리	인접하지 않음
	단 점	장점	장 점
매개변수 전달방식	반드시 스택을 사용	레지스터와 스택 사용	레지스터와 스택 사용
	단 점	장 점	장 점
분기의 특이성	없 음	없 음	파이프라이닝 특성
	단 점	단 점	장 점

4.4 프로세서의 스택관리 형태 특성들의 적용

2.2절에서 언급한 바와 같이 버퍼넘침에 의한 보안침해를 예방하기 위한 방안들이 대부분 소프트웨어 수준에서 연구되고 있기 때문에 불안전하거나 부작용을 동반하고 있다[9-14]. 그러나 4.3 절에서 제시한 것과 같이 프로세서별 특성에 따른 대응방안을 고려하여 프로그램을 작성할 경우 버퍼넘침에 의한 보안침해를 효율적이고 완벽하게 예방할 수 있다. 물론 하드웨어 종속(호환성 결여)라는 문제점이 없는 것은 아니지만, 프로그램 보안성이 심각하게 요구되는 시점에서는 충분한 설득력을 가진다.

이와 같은 프로세서의 특성에 따른 프로그램 작성 지침은 [13, 14]와 같은 컴파일러 수준의 정적 점검도구에 플러그인 형태로 부가되어 현재 사용중인 플랫폼의 유형에 따라 프로그램 작성자에게 적절하게 보내어 질 수 있을 것이다.

5. 결 론

이 논문에서는 버퍼넘침의 개념, 버퍼넘침에 의한 보안침해 과정 및 예방방안에 대한 연구를 소개한 후, 보다 효율적이고 완전한 버퍼넘침 보안침해 예방방안을 제시하였다.

기존에 소개된 대부분의 예방방안은 순수한 소프트웨어 수준에서 접근하고 있어서 불안전하거나 성능저하 등의 부작용을 동반하고 있기 때문에, 여기서는 한 차원 낮은 프로세서의 스택관리 형태의 특성을 고려한 예방방안을 시도하였다. 이를 위하여 특성이 상이한 대표적인 프로세서 세 종류를 집중 분석하였고, 그 결과 그들 프로세서의 스택관리 형태가 버퍼넘침에 영향을 주는 몇 개의 중요한 인자를 지니고 있음을 밝혔다. 그리고 그들 각 영향인자에 따른 버퍼넘침 보안침해에 대한 대응방안을 제시하였다. 이는 곧 프로그램 보안성을 위한 각종 도구에 프로세서 타입에 대한 고려가 필요함을 의미하기도 한다.

본 연구는, 이 번 결과를 이미 개발된 정적 소스코드 검증기에 부가하여 그 효용성을 검증하는 다음 단계로 계속 이어질 예정이다.

참 고 문 헌

[1] Brian W. Kernighan, Dennis M. Ritchie ; 'The C Programming Language', Prentice-Hall., 1978.

[2] Terrence W. Pratt, "Programming Languages : Design and Implementation," PRENTICE-HALL, 1975.

[3] Roger S. Pressman, 'Software Engineering, A Practitioner's Approach', 4th Ed, pp.519-521, McGraw-Hill, 1997.

[4] 이형봉, 박현미, 박정현, "버퍼넘침을 사용한 해킹공격 기법 및 예방 방안", 한국정보처리학회 200추계학술발표논문집(상), pp.129-132.

[5] 박정현, 박현미, 임채호, "소스코드 보안취약점 분석 방법론에 관한 연구", 한국통신정보보호학회 2000종합학술발표회논문집, pp.733-738.

[6] McClure, Scambray, Kurtz, "Hacking Exposed," OSBORNE, 1998.

[7] David A. Wheeler, "Secure Programming for Linux and Unix HOWTO," GFDL, http : //www.dwheeler._com/secure-programs, 2000.

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

[9] Arash Baratloo and Navjot Singh, "Transparent Run- Time Defense Against Stack Smashing Attacks," Bell Labs Research, Lucent Technologies, 2000.

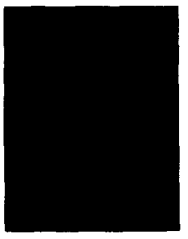
[10] Openwall Project, "Linux kernel patch from the openwall project," http : //www.openwall.com/linux.

[11] Alexandre Snarskii, "Increasing overall security," http : //www.lexa.ru:8100/snar/libparanoia, 1997.

[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] David Evans, "Static detection of dynamic memory errors," Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1996.
[14] Richard Jones, "Bounds checking patches for gcc," http : // web.inter.NL.net/hcc/Haj.Ten.Brugge.
[15] MAURICE J. BACH, "The Design of the UNIX Operating System," PRENTICE-HALL, 1986.
[16] H. M. DEITEL, "An Introduction to Operating Systems," ADDISON-WESLEY, 1983.
[17] Kai Hwang, "Advanced Computer Architecture," McGraw-Hill, 1996.
[18] Intel, Pentium 'Processor Users's Manual(Volume 3 : Architecture and Programming Manual)', Intel, 1993.
[19] Digital Equipment Corporation, "Assembly Language Programmer's Guide," Digital Press, 1996.
[20] SunSoft, "SPARC Assembly Language Reference Manual," Sun Microsystems, 1995.

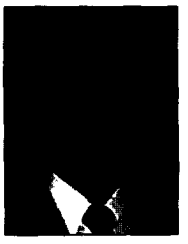
이 형 봉

e-mail : hblee@honam.ac.kr
1984년 서울대학교 계산통계학과 졸업 (이학사)
1986년 서울대학교 계산통계학과 졸업 (이학석사)
1986년~1994년 LG전자 컴퓨터 연구소
1994년~1999년 한국디지탈(주)
1997년~1999년 전자계산조직응용,정보통신기술사
1999년~현재 강원대학교 컴퓨터과학과 박사과정
1999년~현재 호남대학교 정보통신공학부 전임강사
관심분야 : 프로그램언어 및 보안, 운영체제, 멀티미디어 통신



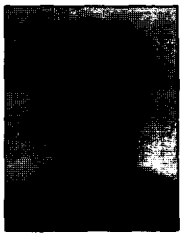
차 홍 준

e-mail : tchahj@kangwon.ac.kr
1964년 춘천교육대학교 초등교육과 졸업
1975년 숭전대학교 전자계산학과 졸업 (학사)
1977년 성균관대학교 전자자료 처리과 졸업 (석사)
1986년 성균관대학교 전산통계학과 졸업(박사)
1978년~현재 강원대학교 컴퓨터과학과 교수
관심분야 : 운영체제, 동적그래픽스, 시스템프로그래밍



노 희 영

e-mail : young@kangwon.ac.kr
1971년 고려대학교 독어독문학과 졸업(학사)
1978년 독일 도르트문트공대 전자계산학 Vor-Diplom과정 수료
1982년 독일 도르트문트공대 전자계산학 Diplom과정 수료
1983년~1984년 한국표준연구소 선임연구원
1984년~현재 강원대학교 컴퓨터과학과 교수
관심분야 : 형식언어 및 자동화이론, 컴파일러 및 프로그램언어, 자연어처리, 소프트웨어공학



이 상 민

e-mail : smrhee@kangwon.ac.kr
1976년 경북대학교 전자공학과 졸업 (학사)
1979년 경북대학교 전자계산학과 졸업 (석사)
1984년 경북대학교 전산통계학과 졸업 (박사)
1985년~현재 강원대학교 컴퓨터과학과 교수
관심분야 : 컴퓨터구조, 디지털시스템설계, 마이크로컴퓨터