Homework 2

51.502: Systems Security

Requirements:

- A Linux (x86) 32-bit Operating System. Install Guest Additions if using VirtualBox. http://sg.releases.ubuntu.com/12.04/ubuntu-12.04.5-desktop-i386.iso.torrent

1. Report on memory protection mechanisms that your platform (OS and compiler) provide.
2. Warm-up 1: Cause the following application to print "ACCESS GRANTED". What is the problem? How to prevent it?

```
#include <stdio.h>

int main(int argc, char *argv[]){
    int arg1, arg2;

    if (argc != 3)
        return -1;

    arg1 = atoi(argv[1]);
    arg2 = atoi(argv[2]);

    if (arg1 < 0 || arg2 < 0)
        return -1;

    if (arg1 + arg2 >= 0)
        return -1;

    printf("ACCESS GRANTED\n");
}
```

3. Warm-up 2: Crash the following application. What is the problem? Fix the application.

```
#include <stdio.h>
  int main(int argc, char *argv[]){
    int i=0;
    char buff[8192];
    char *arg1 =  argv[1];
    if (argc < 2) {
        puts("No arguments");
        return -1;
    }
    while (arg1[i] != '\0'){
        buff[i] = arg1[i];
        i++;
    }
    buff[i] = '\0';
    printf("buff = %s\n", buff);
    return 0;
}
```

4. Warm-up 3: How does the stack look like at the beginning of the `fun3()`'s execution?

```
void fun3(void){
    char a[8];
    double b;
    return;
}
void fun2(void){
    fun3();
}
void fun1(void){
    char a[16];
    int b;
    float c;
    fun2();
}

int main(void){
    fun1();
}
```

5. Buffer Overflow

   There are two types of buffer overflow:

   a) Stack Based Buffer Overflow – Destination buffer resides in stack
   b) Heap Based Buffer Overflow – Destination buffer resides in heap

   Buffer overflow bugs lead to arbitrary code execution. Arbitrary code execution allows attacker to execute his/her own code in order to gain control of the victim machine (e.g. root shell, add a new user, open a network port etc…)

   Compile `vuln1_hw2.c` (see appendix) with the code below. Then find and exploit errors (crashing the application).

```
1 # sudo echo 0 > /proc/sys/kernel/randomize_va_space
2 $ gcc -g -fno-stack-protector -z execstack -o vuln1_hw1 vuln1_hw2.c
3 $ sudo chown root vuln1_hw2
4 $ sudo chgrp root vuln1_hw2
5 $ sudo chmod +s vuln1_hw2
7 $ gdb vuln1_hw2
```

   Line 1: Disables Address Space Layout Randomization (ASLR)
   Line 2: `-fno-stack-protector` → Disables Stack Canary
   `-z execstack` → Disables NX bit

   Use gdb vuln1_hw2 to debug the program. See gdb commands below.

   Modify the code below to overwrite EIP with 'FFFF'. Use info registers to see value of EIP

```
r `python -c 'print "F"*4'`
```

What is the offset from Destination Buffer?

Modify the code to override EBP with 'FFFF' and EIP with 'PPPP'.

Resources:

a) https://en.wikipedia.org/wiki/Stack_buffer_overflow
b) http://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR
c) https://www.cs.cmu.edu/~gilpin/tutorial/
d) http://www.asciitable.com/

6. Integer Overflow

Storing a value greater than maximum supported value is called integer overflow. Integer overflow on its own doesn't lead to arbitrary code execution, but an integer overflow might lead to stack overflow or heap overflow which could result in arbitrary code execution.

Data types size and its range:

Data Type	Size	Unsigned Range	Signed Range
char	1	0 to 255	-128 to 127
short	2	0 to 65535	-32768 to 32767
int	4	0 to 4294967296	-2147483648 to 2147483647

When we try to store a value greater than maximum supported value, our value gets wrapped around. For example, when we try to store 2147483648 to signed int data type, its gets wrapped around and stored as -21471483648. This is called integer overflow and this overflow could lead to arbitrary code execution.

Compile vuln2_hw2.c with the code below. Then find and exploit errors (crashing the application).

```
1 # sudo echo 0 > /proc/sys/kernel/randomize_va_space
2 $ gcc -g -fno-stack-protector -z execstack -o vuln2_hw1 vuln2_hw2.c
3 $ sudo chown root vuln2_hw2
4 $ sudo chgrp root vuln2_hw2
5 $ sudo chmod +s vuln2_hw2
7 $ gdb vuln2_hw2
```

Use `gdb vuln2_hw2` to debug the program. See `gdb` commands below.

Modify the code below to overwrite EIP with 'FFFF'. Use info registers to see value of EIP

```
r `python -c 'print "F"*4'`
```

What is the offset from Destination Buffer?

Modify the code to overwrite EBP with ‘FFFF’ and EIP with ‘PPPP’ and the remaining space with `W’s. Use `x/80x ($esp-28)` to view the remaining space.

7. Hand in. If submitting an image, image of desktop must be shown. Do not crop.
    a. Warm-ups
        i. Short report per exercise (+ a screenshot and fix description -- if applicable)
    b. Buffer Overflow.
        i. code which crashed `vuln1_hw2`.
        ii. EIP overwritten with ‘FFFF’
        iii. What is the offset from Destination Buffer?
        iv. EBP overwritten with ‘FFFF’ and EIP are overwritten with ‘PPPP’
    c. Integer Overflow
        i. code which crashed `vuln2_hw2`.
        ii. EIP overwritten with ‘FFFF’
        iii. What is the offset from Destination Buffer?
        iv. EBP overwritten with ‘FFFF’ and EIP are overwritten with ‘PPPP’ and remaining space with W’s.

GDB commands

```
# show debugging symbols
list main

# show the assembly code
disas main

# run the program, with input
r Hello
r `python -c 'print "A"*3'`

# confirm overwrite of ebp register
info registers

# examine memory address
x/200x ($esp - 550)

# show value of Instruction Pointer Register (EIP)
p/x $eip
```

Appendix.

vuln1_hw2.c:

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

int main(int argc, char* argv[]) {
        char buf[128];
        strcpy(buf,argv[1]);
        printf("Input:%s\n",buf);
        return 0;
}
```

vuln2_hw2.c:

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

void validate_passwd(char* passwd) {
 char passwd_buf[7];
 unsigned char passwd_len = strlen(passwd); /* [1] */
 if(passwd_len >= 4 && passwd_len <= 8) { /* [2] */
  printf("Valid Password\n"); /* [3] */
  fflush(stdout);
  strcpy(passwd_buf,passwd); /* [4] */
 } else {
  printf("Invalid Password\n"); /* [5] */
  fflush(stdout);
 }

int main(int argc, char* argv[]) {
 if(argc!=2) {
  printf("Usage Error:   \n");
  fflush(stdout);
  exit(-1);
 }
 validate_passwd(argv[1]);
 return 0;
}
```