**Introduction Information Security**

**Project #1 Buffer Overflow**

**Goals:**

* Understanding the concepts of buffer overflow
* Exploiting a stack buffer overflow vulnerability
* Exploring how to mitigate code reuse attacks (advanced buffer overflow attacks)

Students should be able to clearly explain: 1) what buffer overflow is; 2) why buffer overflow is dangerous; and 3) how to exploit a buffer overflow. Students are expected to launch an attack that exploits a stack buffer overflow vulnerability in the provided toy program. Finally, students are encouraged to explore the mitigation of code reuse attacks. Some research papers are provided for reading.

There are two tasks in this project.

**Task 1. Understanding Buffer Overflow**

**1) Stack buffer overflow** (40 points)

Write a program that contains a stack buffer overflow vulnerability. Show what the stack layout looks like and explain how to exploit it. You are not required to write the real exploit code, but you may want to use some figures to make your description clear.

**Deliverable:** a pdf file containing your vulnerable program (paste your code in the pdf directly) and your explanation.

**Task 2. Exploiting Buffer Overflow (**60 points**)**

The following C code contains a stack buffer overflow vulnerability. Please write an exploit (e.g., Python script) to open a shell on Linux. The high level idea is to overwrite the return address with the address of function *system()*, and pass the parameter *“sh”* to this function. Once the return instruction is executed, this function will be called to open a shell.

Steps

1) Install a virtual machine with 32-bit Ubuntu 14.04.03 LTS (<http://www.ubuntu.com/download/desktop/>), if you do not have one. We choose 32-bit because it is MUCH easier for exploiting (on x86\_64, attacker has to use return-oriented programming to load parameters from memory to registers).

2) Disable ASLR: *echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space.* You should enable ASLR back after your project by*: echo 2 | sudo tee /proc/sys/kernel/randomize\_va\_space*

3) Compile the provided C code: *gcc sort.c -o sort -fno-stack-protector.*

4)To run this program, put some hexadecimal integers in the file: *data.txt*, and execute *sort* by: *./sort data.txt*

5) When you put a very long list of integers in *data.txt*, you will notice *sort* crashes with memory error segfault, this is because the return address has been overwritten by your data.

6) Now you can craft your shellcode in *data.txt*. Again, your goal is to overwrite the return address with the address of function “system()” and pass it with the address of string “sh”.

7) Have fun.

Deliverable: either an exploit script (any scripting language is allowed, but your script must contain everything, so that the TA can directly run it by *“./script”*) or the *data.txt* file you craft.

sort.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\*

\* a toy program for learning stack buffer

\* overflow exploiting

\* It reads a list of hex data from the

\* specified file, and performs bubble sorting

\*/

long n = 0, c = 0, d = 0, swap = 0;

FILE \*fp = NULL;

void bubble\_sort()

{

long array[10];

// loading data to array

printf("Source list:\n");

char line[sizeof(long) \* 2 + 1] = {0};

while(fgets(line, sizeof(line), fp)) {

if (strlen((char \*)line) > 1) {

sscanf(line, "%lx", &(array[n]));

printf("0x%lx\n", array[n]);

++n;

}

}

fclose(fp);

// do bubble sorting

for (c = 0 ; c < ( n - 1 ); c++)

{

for (d = 0 ; d < n - c - 1; d++)

{

if (array[d] > array[d+1])

{

swap = array[d];

array[d] = array[d+1];

array[d+1] = swap;

}

}

}

// output sorting result

printf("\nSorted list in ascending order:\n");

for ( c = 0 ; c < n ; c++ )

printf("%lx\n", array[c]);

}

int main(int argc, char \*\*argv)

{

if(argc!=2)

{

printf("Usage: ./sort file\_name\n");

return -1;

}

fp = fopen(argv[1], "rb");

bubble\_sort();

return 0;

}

A sample of *data.txt*. Craft your shellcode in *data.txt*

1

3

5

7

80

a

d0

**Deliverable:** put your answer in the same pdf file of task 1. That is, the final project deliverables are: one pdf file + one script/data.txt