Austin Peterson
UIN: 926006358

Buffer Overflow Attack

A buffer overflow vulnerability occurs when an input buffer has data written into it from a buffer or a file without checking the input bounds or using other anti-overflow defensive measures. This allows an attacker to enter specific code that overwrites fields outside of the buffer variable, including the return address, with code that hijacks the operation of the program and often spawns a shell program, giving the attacker a direct line of connection and control with the host system.

For this assignment, several steps had to be taken at first to allow for a basic overflow exploit. First, address randomization was turned off, done using " the sysctl command "#sysctl -w kernel.randomize_va_space=0" run as root. The exploit was compiled normally, however the stack program with the vulnerability had to have stack protector turned off and executable stack enabled, done during compilation with the compile arguments "-z execstack -fno-stack-protector". Disabling stack protector allowed us to write data outside the bounds of the original buffer, an obvious step during a buffer overflow attack. Since this attack involved running executable code inject into the stack, we must enable stack execution on the OS, otherwise the data within the stack would simply have rw- access and our shellcode would not run.

Once our runtime was set up, we were able overflow the 18 byte char buffer with data from a custom written file containing shell code and a return address that allowed us to gain control of the stack execution and spawn a shell. The process for this is simple in theory: Create a buffer larger than the buffer being written into, create a NOP sled to increase the odds of success, copy the shellcode somewhere in he custom buffer, usually at the beginning or the end, and set a custom return address that fall somewhere in our NOP sled so that eventually our code will run. If the input buffer were large enough then we could easily inject our shellcode in the beginning of the buffer and set the return address to the address of the buffer, however our shellcode was too large for the buffer that we were given. The solution to this program is to write our shellcode at the end of the buffer and after a substantial NOP sled. This allows some more wiggle room with finding an appropriate return address, which is the hardest part of this exploit. If the return address is outside of the NOP sled then a seg fault gets thrown and the attack fails, however if it lands in the NOP sled then those commands will run until eventually the shellcode is reached and executed.

Tasks 1 & 2:

The hardest part of this assignment was getting the base exploit to run, which included creating a NOP sled, rewriting the return address to somewhere in that sled, and inserting our assembly shellcode to the end of our buffer. My exploit.c takes runtime arguments for the buffersize and the offset so that any changed wouldn't have to be changed in the code itself. The return address is created using the get_sp(void) function that was suggested in the provided documentation. The return value from that function was increased by the input variable offset, which would ideally land that address somewhere in the NOP sled. The hardest part, in my opinion, was finding the correct location to write the custom return address to. In the end the solution was to add an offset of size 0x1A to the address of the buffer, which included space for the 18 byte

Austin Peterson
UIN: 926006358

char array and the 8 bit base pointer address. Another solution that I played around with was simply adding an offset to the base pointer address, found similarly to the stack pointer using in calculating the return address, since the ebp is located right next to the return address on the stack. Unfortunately, I didn't have much luck with that solution. Next, loops are used to copy the return address at the buffer+offset pointer and to copy the shellcode into the end of the NOP sled. After a '\0' terminal character is put into the end of the buffer, the contents of the custom buffer are written to the badfile to exploit the stack executable. Initially I had planned on copying data into the buffer using memcpy or strncpy, however I had more luck using loops so I ended up going with that. The first terminal window shows the two terminal windows show the shells from parts 1 and 2. Part two included adding additional asm code for a set-uid call that elevated the permission of the program, giving root access.

```
*/

//Function returns the stack pointer
unsigned long get_sp(void) {
        __asm__("movl %esp,%eax");
}

unsigned long get_bp(void){
    __asm("movl %ebp, %eax");
}

void main(int argc, char**argv){
        int buffer_size = 517;
        int offset = 0;
        FILE* badfile;

        //arg1 = buffersize, arg2 = offset
        if(argc = 2) { buffer_size = atoi(argv[1]); }
        if(argc =3) { buffer_size = atoi(argv[1]); offset = atoi(argv[2]); }

        char buffer[buffer_size];
        memset(&buffer,0x90, buffer_size);

        //*((long*)buffer + 0x36) =  0xbfffeb9e + 0x128;
        long* return_addr = get_sp() + offset;

        printf("\nUsing address 0x%x\nBuffer size: %i\nOffset: %i\n", get_sp(), buffer_size, offset);

        long* ptr_addr = (long*)(buffer + 0x1A); //offset by 26 bytes, 18 for char[18], 8 for ebp
        //long* ptr_addr = get_bp() + 0x04;

        for(int i=0; i<8; i++) {  //insert return address
                *(ptr_addr++) = return_addr;
        }
        //memcpy(ptr_addr, return_addr, 8);

        for(int i=0; i<sizeof(code); i++) {
                buffer[buffer_size - (sizeof(code) +1) +i] = code[i]; //copy the shellcode into the bu
        }

        buffer[buffer_size -1 ] = '\0'; //null terminator
        printf("Buffer created, writing to badfile\n");

        //Save contents to the file "badfile"
        badfile = fopen("badfile", "w");
        fwrite(buffer, 517,1,badfile);
        fclose(badfile);
}
```

```
./exploit 517 200

Using address 0xbfffea78
Buffer size: 517
Offset: 200
Buffer created, writing to badfile
gcc -o stack -fno-stack-protector -z execstack stack.c
./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

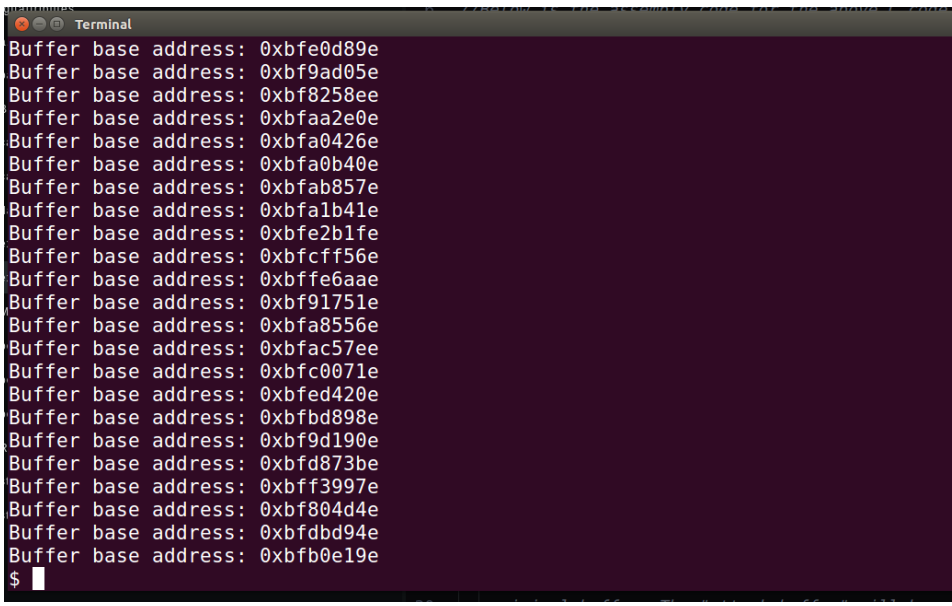
```
./exploit 517 200

Using address 0xbfffeae8
Buffer size: 517
Offset: 200
Buffer created, writing to badfile
gcc -o stack -z execstack -fno-stack-protector stack.c
./stack
Buffer base address: 0xbfffeb2e
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Austin Peterson
UIN: 926006358

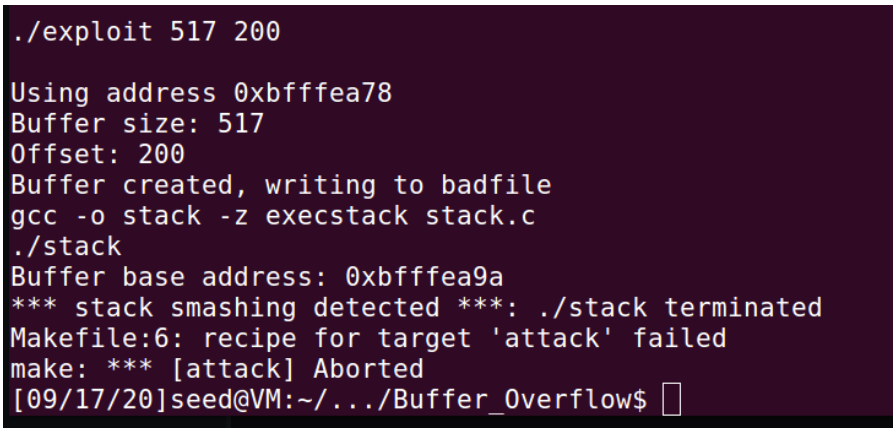
Task 3:

Task 3 was related to address randomization, specifically running our exploit with address randomization turned on. Where this gets more complicated is that with address randomization turned on, placing the return address in the correct location, becomes much more difficult. Without it, $esp has less variety and we have a much easier time consistently gaining access. While address randomization does make the exploitation process more difficult, we were still able to gain access using our badfile overflow attack. Using the provided loop, $ sh -c "while [1]; do ./stack; done;", we were able to run the vulnerable program infinitely until the addresses were compatible and our exploit ran successfully. This was not immediate, but it was much faster than manually running the stack program. The results are shown below. This approach did get around the randomization issue, however I'm not sure how "sneaky" it is realistically. Running a program this many times, assuming we have that kind of access in the first place, would like be conspicuous and would not go unnoticed.



Task 4:

Task 4 pertains to the stack guard, which monitors and enforces the bounds on buffers. Overwriting is the whole point of a buffer overflow attack; this is obviously an important aspect of the vulnerable program. When compiling and running without the *-fno-stack-protector* option, we get an error saying that stack smashing was detected and the program aborts. This was no surprise since stack smashing is the base concept of this project.



Austin Peterson
UIN: 926006358

Task 5:

Task 5 is all about the executable stack option which allows our shellcode to run in the overwritten stack space. When the vulnerable stack program was compiled with *-z noexecstack* instead of *-z execstack* we were not able to get a running shell and were instead greeted with the all too familiar segmentation fault. Since any data found in the stack is not seen as executable, the shellcode/NOP is seen as invalid data, like how running an executable in the linux file system throws an error without +x permission. While there are ways to get around a non-executable stack, it makes the process more difficult as injected code would have to be injected to the heap or statically with a rerouted return address, which can be difficult to calculate.