Alex Miller
Assignment 8 Writeup

sploit0.c
My attack works by feeding a string of length 100 into target0; bytes 20-23 contain the address following the return address of the stack frame of bad_echo(); this points to where the shellcode will find its place in memory (following the overflow). This payload, when copied to 16-byte buf with strcpy(), overflows the return address of foo() so that it points to the start of the shellcode. Therefore, when foo() returns, EIP points to the start of the shellcode; this executes the shellcode.

sploit1.c
My attack works by feeding a 512-byte string into target1; bytes 260-263 contain the address following the return address of the stack frame of foo(); this points to where the shellcode will find its place in memory (following the overflow). This payload, when copied to 256-byte buf with strcpy(), overflows the return address of foo() so that it points to the start of the shellcode. Therefore, when bar() returns, foo() will return and then restore EIP so that it points to the start of the shellcode; this executes the shellcode.

sploit2.c
The tricky part of this attack is that the function nstrcpy() restricts the amount of data we can overwrite to the target buffer (buf). It does, however, make one mistake; it allows us to overflow one more byte of data than the size of buf. We can, therefore, overflow the end of the saved EBP address to some address within the buffer. The payload we feed into taregt2, then, is a 201-byte string; the string is filled with NOP bytes, contains the shellcode somewhere in the middle of it, and is ended by an address pointing to the beginning of the buffer and end-capped by a byte that, when it overflows the last byte of the saved ebp, makes the ebp point to an address in memory two words prior to where it is stored. Therefore, when buf is overflowed with our input, the saved ebp makes us return to the start of the buffer, which executes the shellcode.

sploit3.c
The mistake the target file makes is not using consistent typing when checking the qualities of the input; the input_size is an int when compared to the length of the input and a short when comparing it to the number 399. We can, therefore, feed the program a 65935-byte string and an int of value 65935 as input. This stops the condition ‘real_size > input_size’ from evaluating to true; it also makes ‘(short) input_size != 399’ evaluate as false, since input_size interpreted as a short represents a value of 399. We can, therefore, make a successful attack by feeding a string of length 65935 into target3; bytes 404-407 contain the address following the return address of the stack frame of foo(); this points to where the shellcode will find its place in memory (following the overflow). This payload, when copied

to 400-byte buf with strcpy(), overflows the return address of foo() so that it points to the start of the shellcode. Therefore, when foo() returns, EIP points to the start of the shellcode; this executes the shellcode.

sploit4.c

When target4 reads in its input; it first reads the value that becomes count as an unsigned long and then converts it into an int. This allows us to feed the program a large value that assigns count a negative value; therefore 'count < MAX_WIDGETS' always evaluates to true. Moreover, we can further control the value returned by 'count * sizeof(struct widget_t).' Since sizeof is an operator that returns a size_t variable (an unsigned int), when 'sizeof(struct widget_t)' is multiplied by count (an int), it returns an unsigned int that overflows. Therefore, we can make 'count * sizeof(struct widget_t)' evaluate to a desired result (x) by making count = $2^{31} + (x / 20)$. Working backward then; we want to overflow buf with a string of 20100-bytes; bytes 20004-20007 contain the address following the return address of the stack frame of foo(). This address points to where the start of shellcode will end up after the overflow (which starts at byte 20008 of the payload). We then add to the front of this payload the string "2147484653," which is our desired count variable ($2^{31} + 1005$) followed by a comma. This total input, when passed to target4, creates a shell.

sploit5.c

Before we begin our attack, we need three things:

1. The address of the system call function for libc (SYS_ADDR)
2. The address of the exit function in libc (EXIT_ADDR)
3. The address of the address of the string "/bin/sh" in libc (SH_ADDR)

The idea of this attack is to overflow the stack frame of foo() to simulate the stack frame of launching a shell. Therefore, if this is the stack frame of foo() prior to our overflow:

[name][ebp][ret]....

We want our overflow to result in this:

{XXXX][XXXX][SYS_ADDR][EXIT_ADDR][SH_ADDR]

Where X is any byte. This makes it so that, when foo() returns, it executes a shell just as if it had been called by the original program.
We can do this by feeding target5 input that looks something like:
/0x90/0x90/0x90/0x90/0x90/0x90/0x90/0x90/0x90/0x90SYS_ADDREXIT_ADDRSH_ADDR

Sources

https://uchicago.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=c45c301d-c6f2-4ede-b0bc-ab760149b963
https://www.welivesecurity.com/2016/05/10/exploiting-1-byte-buffer-overflows/

http://phrack.org/issues/49/14.html#article
https://study.com/academy/lesson/binary-division-multiplication-rules-examples.html
http://www.cplusplus.com/reference/climits/