Jack Bauer

PUID: 0030567277

A. Vulnerable1
   1. This program takes in a single command line argument. This argument is then passed to the call_vul() function which has one local variable: a character array, temp, of length 50. The call_vul() function then calls the vulnerable() function and passes it the original command line argument. The vulnerable() function has one local variable: a character array, buf, of length 100. Lastly, the vulnerable() function uses strcpy() to copy the command line argument into the buf character array.
   2. The vulnerability in this program is when the vulnerable() function calls strcpy() on line 7. The buf character array is of size 100, but the program will take in a string of any size and copy it to buf. If the input is larger than the size of buf, memory locations that don't belong to buf, such as the return address of vulnerable(), could be overwritten resulting in unexpected behavior.
   3. The goal for this exploit is to overwrite the return address of the vulnerable() function to the location of shellcode, thus launching a shell with root permissions. To do so, I create an input string comprised of the shellcode, followed by 59 "a" characters, then the address "0xbffebdac" which is the address that I inserted the shellcode into. By inserting this string into the buf character array, the overflow will reach the location of the return address of the vulnerable() function and overwrite it. Therefore, the return address will point to the address containing the shellcode which, when executed, will launch a shell with root permissions.
   4. See submitted code
   5. One thing that can prevent a buffer overflow from happening in this program specifically is to check the length of the command line argument to ensure that it is less than or equal to 100 (i.e. the length of the buf character array). In a general sense, using strncpy() instead of strcpy() forces the program to copy a specific number of characters, rather than an arbitrary amount, to ensure that the array will not overflow.

B. Vulnerable2
   1. This program takes in a single command line argument which is a filename that contains a 32-bit count followed by a series of 32-bit integers. The filename is then passed on to the call_vul() function that contains a length 66 character array and a call to the read_file() function. Call_vul() passes on the command line argument to read_file() which then opens the file, reads the first 32-bit integer (the count), and allocates an unsigned integer pointer named buf based on the count multiplied by the size of an unsigned integer (4). The read_file() function then passes on the file object, the integer pointer buf, and the count from the file to the read_elements() function. The read_elements() function reads through each 32-bit integer in the file and places them in the unsigned integer pointer buf.
   2. The vulnerability in this program is in line 26 where the unsigned integer pointer buf is allocated. If a very large value is used for count and it is then multiplied by 4, the result will wrap around to a small integer value causing a relatively small buf to be allocated. The read_elements() function reads through the file based on the count variable, so if count is a very large number, it has the potential to read more data than what buf can store, consequently causing an overflow.
   3. The goal is to provide a very large value for count so when it is multiplied by 4, the result will wrap around to a smaller value. This way, the unsigned integer pointer buf will have a relatively

small amount of memory allocated to it. Then, we have the opportunity to write more data to buf than it has memory allocated. I create a file with a string of containing the count value 0x40000009 (slightly larger than ¼ of the maximum integer value) followed by the shellcode, 39 bytes of NOPs (0x90), and the address 0xbffebdb0 which is the address that holds the beginning of the shellcode. The idea is to have a count value that is greater than the amount of space allocated for buf so the program is able to read more data than can fit inside buf's allocated memory. This will cause an overflow that will allow the program to reach the return address of read_file() and be overwritten by the 0xbffebdb0, the address containing the beginning of the shellcode.

4. See submitted code
5. In this program specifically, this vulnerability can be avoided by restricting the read_elements() function to read in data from the file based on the amount of memory allocated to buf rather than reading in an amount of data based on the count variable. This will ensure that no data read from the file will be stored outside of buf's allocated memory space. In a general sense, constants should be set that are equal to the maximum/minimum values integers can hold. When performing operations on integers, the resulting values of these operations should be checked against the constants to ensure they are not outside of their expected ranges.

C. Vulnerable3

1. This program takes in a single command line argument and passes that argument to the vulnerable() function. The vulnerable() function then copies that argument, using strcpy(), to a character array of length 1024 named buf. The difference between this program and vulnerable1, though, is that the stack position is randomly offset by 0-255 bytes each time it is run.
2. The vulnerability in this program is when the vulnerable() function calls strcpy() on line 10. The buf character array is of size 1024, but the program will take in a string of any size and copy it to buf. If the input is larger than the size of buf, memory locations that don't belong to buf, such as the return address of vulnerable(), could be overwritten resulting in unexpected behavior.
3. The goal for this exploit is to overwrite the return address of the vulnerable() function to the location of shellcode, thus launching a shell with root permissions. To do this, I first had to run the program many times to find an educated estimate of the range of addresses where buf is stored as well as how far away the return address of vulnerable() is relative to the last allocated space of buf. I was able to determine that the address 0xbffebb01 (chosen after much trial and error) would always be allocated for buf. I formatted a string to fill the first ¾ of buf's allocated space with NOP instructions (0x90), followed by the shell code, 3 more NOP instructions, and finally the address 0xbffebb01 repeated 56 times which resulted in a buffer overflow. The idea is to fill up the majority of the buffer with NOP instructions so there will be a specific address that will always contain a NOP instruction (in this case 0xbffebb01). The three extra NOP instructions were included because I found there was an address after the shellcode that was stored slightly offset of what I wanted. This resulted in the repeated 0xbffebb01 addresses to be stored incorrectly. I repeated 0xbffebb01 56 times because I found that after storing the NOP instructions and the shellcode, adding 56 more 4 byte memory addresses would always result in reaching the return address of vulnerable(). Since I knew from testing that 0xbffebb01 would always contain a NOP instruction, I overwrote the return address of vulnerable() to point to that address. Once the return address pointed to 0xbffebb01, the program would skip all of the NOP

instructions and immediately execute the next viable memory space, which was the shellcode. Thus, after skipping all of the NOP instructions, a shell with root permissions was opened.

4. See submitted code
5. One thing that can prevent a buffer overflow from happening in this program specifically is to check the length of the command line argument to ensure that it is less than or equal to 1024 (i.e. the length of the buf character array). In a general sense, using strncpy() instead of strcpy() forces the program to copy a specific number of characters, rather than an arbitrary amount, to ensure that the array will not overflow.