# 1 Sploit 1

The target uses the function "strcpy". When looking at its prototype, which is char *strcpy(char *dest, const char *src), we can see that there is no actual check for bounds, and, since the function terminates copying the string as soon as it encouters a null character, if the source is bigger than the destination then the remaining bytes will overflow in the memory and will possibly override important areas of this memory.

The target copies the input argument argv[1] into a buffer of "char" (1 byte) of size 240. Since the buffer "buf" is created in the function "foo", we will set a breakpoint at line 15 which is just after the execution of the "bar" function, where the buffer is filled with "strcpy". We do that to actually retrieve the address in memory where the buffer starts (at which address is located the first element of the buffer). After reaching this breakpoint when executing the code using GDB, we type the command "i r" to get the location of the ESP and EBP pointers. The ESP pointer is the current stack pointer (current top of the stack) actually pointing somewhere in the "foo" stack frame, and the EBP pointer is the base pointer for the current stack frame, and then points to the last address of the "foo" stack frame. When we call a function, the return address will always be 4 bytes after the base pointer (at EBP + 4). This means that copying more than 240 elements to "buf" will lead to a buffer overflow and then allow us to change the return address of the "foo" function. The command "i r" gives ESP = 0xbffffc70 and EBP = 0xbffffd78. This means the content of the buffer is located between these two addresses, and the return address (that we then want to modify) is located at 0xbffffd7c. We can get the address where is located the first element of the buffer "buf" by entering the command "x/100x buf", and we can see that the buffer starts at 0xbffffc88. Notice that after the last element of the buffer, we should be at the location of the EBP pointer, which is the case since 0xbffffd78 - 0xbffffc88 = 0xf0 = 240 which is indeed the number of allocated elements (240 elements of 1 bytes) for the buffer.

We will then submit as input (and source for the "strcpy" function) an array of 249 1-byte characters. The last byte (at position 248 of the array) will be the NULL byte " so that "strcpy" will stop copying immediately after having overwritten the return address. Indeed, the new return address (that will be the one sending the program to the location of our shellcode) will be stored at the positions 244, 245, 246 and 247 of our input array (the first position is the position 0). We wrote the content of our shellcode array (of length 45) at the beginning of the buffer. The remaining elements (position 45 to 239) will be anything (we chose to put 'f' characters, which each will be displayed as "66" since their ASCII hexadecimal representation is 0x66) and will be written in the remaining 195 slots of the buffer "buf". The elements at position 240, 241, 242, 243 will also be "f" and will overwrite the value pointed by EBP. Positions 244, 245, 246 and 247 of the input array are then the ones we have to focus on. Since we decided to write the content of our "shellcode" array starting from the address 0xbffffc88 (beginning of the buffer), this means we will have input[244] = '88', input[245] = ", input[246] = " and input[247] = " since we are working

in a little-endian fashion, and so the return address will be 0xbffffc88 which will then yield our shell.

## 2 sploit2

The target actually writes the given input argument into a buffer, but if the input argument is longer than the size of the buffer, it will only copy the first 241 (1-byte) elements of the input argument. But since the target code allocated only 240 1-bytes elements to the buffer, we can exploit this code via a vulnerability called the "Off-By-One Overflow". When the source string length is equal to the destination buffer length, a single NULL byte can get copied just above the destination buffer. Here, since the destination buffer is located in the stack, the single NULL byte could overwrite the least significant byte of caller's EBP stored in the stack and this could lead to arbitrary code execution.

The buffer "buf" is created in the function "bar", so we will set a breakpoint at line 22 which is just after the execution of the "nstrcpy" function, where the buffer is filled. We do that to actually retrieve the address in memory where the buffer starts (at which address is located the first element of the buffer). After reaching this breakpoint when executing the code using GDB, we type the command "i r" to get the location of the ESP and EBP pointers. The ESP pointer is the current stack pointer (current top of the stack) actually pointing somewhere in the "bar" stack frame, and the EBP pointer is the base pointer for the current stack frame, and then points to the last address of the "bar" stack frame. The command "i r" gives ESP = 0xbffffc70 and EBP = 0xbffffd78. This means the content of the buffer is located between these two addresses. We can get the address where is located the first element of the buffer "buf" by entering the command "x/100x buf", and we can see that the buffer starts at 0xbffffc88. Notice that after the last element of the buffer, we should be at the location of the EBP pointer, which is the case since 0xbffffd78 - 0xbffffc88 = 0xf0 = 240 which is indeed the number of allocated elements (240 elements of 1 bytes) for the buffer. This means that copying 241 elements to "buf" will lead to a buffer overflow and then allow us to change the least significant byte of the value pointed by the EBP of the "bar" stack frame.

We will then submit as input an array of 241 1-byte characters. The last byte (at position 240 of the array) will be the NULL byte " so that the value pointed by EBP will now become 0xbffffd00. Since the first element of the buffer is located at 0xbffffc88, the address 0xbffffd00 is part of destination buffer "buf" and then is a stack location over which an attacker has control, and so can achieve arbitrary code execution since he has then control over the instruction pointer EIP (contains the address of next instruction to be executed). After the EBP of "bar" gets overwritten by the function "nstrcpy" and returning to the next "bar" instruction, since the latter is a "leave" according to the command "disassemble bar", it will unwind this function's stack space and restores EBP, and so ESP will become at this moment EBP + 4. The following instruction is "ret", still according to "disassemble bar" command, and the program will

then return to the instruction located at the current ESP. By overwritting EBP least significant byte by the NULL byte, ESP will point at 0xbffffd04, which must then contain the return address to our shellcode. To find the offset to the location (0xbffffd04) of this new return address from the beginning of the buffer (0xbffffc88), we have the following calculation : 0xbffffd04 – 0xbffffc88 = 0x7c = 124. Then this means the new return address (that will be the one sending the program to the location of our shellcode) will be stored at the positions 124, 125, 126 and 127 of our input array (the first position is the position 0). We then have to write the content of our "shellcode" array in the buffer starting at this return address. We decided to write the content of our "shellcode" array starting from the address 0xbffffd48 (so at position 192 of the input array), then this means we will have input[124] = '48', input[125] = '', input[126] = '' and input[127] = '' since we are working in a little-endian fashion and so the return address will be 0xbffffd48 which will then yield our shell. The other elements of the input array will be anything (we chose to put 'f' characters, which each will be displayed as "66" since their ASCII hexadecimal representation is 0x66).

# 3 Sploit 3

The target uses the function "memcpy". The function's signature is void *memcpy(void *str1, const void *str2, size-t n) and then copies n characters from memory area str2 to memory area str1. This means we can control the bounds, and, since the function terminates only when n characters have been copied, if n is sufficiently large, and if the source input is bigger than the destination buffer, the remaining bytes overflowing in the memory will override important areas of this memory.

The target retrieves via the function "strtoul" an integer "count" and an array of characters. It then creates a buffer of "struct widget-t" and allocates 240 slots to it. Since a "struct widget-t" element is 20 bytes long, 4800 bytes has been allocated to the buffer. It then copies count * sizeof(struct widget-t) bytes from the array of character retrieved via "strtoul" (it then copies the count * 20 first bytes). Since the buffer "buf" is created in the function "foo", we will set a breakpoint at line 20 which is just after the execution of the "memcpy" function, where the buffer is filled. We do that to actually retrieve the address in memory where the buffer starts (at which address is located the first element of the buffer). After reaching this breakpoint when executing the code using GDB, we type the command "i r" to get the location of the ESP and EBP pointers. The ESP pointer is the current stack pointer (current top of the stack) actually pointing somewhere in the "foo" stack frame, and the EBP pointer is the base pointer for the current stack frame, and then points to the last address of the "foo" stack frame. When we call a function, the return address will always be 4 bytes after the base pointer (at EBP + 4). This means that copying more than 4800 bytes to "buf" will lead to a buffer overflow and then allow us to change the return address of the "foo" function. The command "i r" gives ESP = 0xbfffd8b0 and EBP = 0xbfffeb88. This means the content of the buffer is

located between these two addresses, and the return address (that we then want to modify) is located at 0xbfffeb8c. We can get the address where is located the first element of the buffer "buf" by entering the command "x/100x buf", and we can see that the buffer starts at 0xbfffd8c8. Notice that after the last element of the buffer, we should be at the location of the EBP pointer, which is the case since 0xbfffeb88 - 0xbfffd8c8 = 0x12c0 = 4800 which is indeed the number of allocated elements (4800 elements of 1 bytes, or 240 elements of 20 bytes) for the buffer. A solution could then have been to give a "count" value so that count * 20 equals 4808 and overwrite the return address with the bytes at position 4804, 4805, 4806 and 4807 (with first byte of the array of character at position 0) of the array of characters after the comma in the input argument. Those 4 bytes would represent the chosen return address where our shellcode would be located.

The problem is the "if" statement : we can't give to "count" a value greater than 239. This means, "memcpy" would copy only the first 4780 bytes of our array of characters, which is not enough to reach and overwrite the return address. A solution would be to produce an overflow when calculating count * 20 (the number of bytes to copy) so that it gives a value higher than 4780 but with a count value lower than 240. The only way is to give "count" a very high negative value so that it passes the "if" statement and when calculating count * 20 the minimum possible value for an integer in a computer program (-2147483648) is exceeded, which would then possibly give a positive value. With count = -214748124 we obtain count * 20 = 4818 which is the minimum possible acceptable value for the number of bytes to write with "memcpy". We want to write 4808 characters (the part after the comma following the input format) and "-214748124," is 11 bytes (11 characters long), so we create an input array of 4819 bytes to store everything we need to overwrite the return address and yield our shellcode. The characters that will be written at the first position of "buf" is at position 11 of the input array (because the first 11 characters of the input array are "-214748124,"), and we actually decided to write the content of our "shellcode" array starting from the address 0xbfffeb58, which is 0xbfffeb58 - 0xbfffd8c8 = 0x1290 = 4752 bytes after the beginning of the buffer. This means the content of the "shellcode" array will be written from position 4752 + 11 = 4763 of the input array. Finally, since the return address we want to overwrite is located 4804 bytes after the beginning of the buffer, we will write the 4 bytes (of the new return address) starting from position 4804 + 11 = 4815 of the input array. We want the return address to be 0xbfffeb58 so we will have input[4815] = '58', input[4816] = ", input[4817] = " and input[4818] = " since we are working in a little-endian fashion, and so the return address will be 0xbffffd58 which will then yield our shell. The other elements of the input array will be anything (we chose to put 'f' characters, which each will be displayed as "66" since their ASCII hexadecimal representation is 0x66).