

Project 1: Control Hijacking

Julian Cooper

AA228/CS238, Stanford University

JELC@STANFORD.EDU

1. Buffer Overflow

Vulnerability: Want to exploit `strcpy` being called without bounds checking.

Exploit: We can overwrite the return address of `foo` with the address of our shellcode, then execute the shellcode to get a root shell.

Working notes:

- Using `gdb` we can find address of the buffer variable, `0x7ffffffdc20`.
- We then can find the address of the return address, `0x7ffffffdd28`.
- The difference of these two addresses is `0x108` (264), which is the number of bytes we need to overwrite to get to the return address. Note, we need to add 8 bytes to this to account for the saved return address, so our exploit must be 272 bytes long.
- Copy shellcode without terminating null pointer into our buffer exploit.

2. Off-by-One

Vulnerability: Want to exploit improper for loop construction which writes 129 bytes to a 128 byte static array.

Exploit: We can overwrite the last byte of "previous base pointer" address which sits right above our `char* input` local variable in the `foo` stack frame. We can use this to shift our prev `rbp` down by 16 bytes, which prompts the program to return the address for `char* input` and execute arbitrary code we have passed in that argument.

Working notes:

- Using `gdb` we can find address of the input variable in the `foo` stack frame, `0x7ffffffeb08`.
- We then can find the address of the previous `rbp` in the `foo` stack frame, `0x7ffffffeb10`.
- The difference of these two addresses is `0x08` (8 bytes), which makes sense, this is just the memory allocated for our base pointer address.
- Idea: want to overwrite the last byte of the previous `rbp` address with `0x00` (replaces `0x10`), which will shift the `rbp` down by 16 bytes (since we are in base 16) and cause our program to use the address of our input variable as the return address and execute whatever code is stored in that memory location.

- Similarly, to exploit 1, we can then just copy our shell code into the input variable (without the terminating null pointer to avoid early exit) and execute it.
- Note, our shellcode is only 23 bytes and so written to the bottom of our input variable (exploit). Since our exploit needs to be 129 bytes long, we fill the rest with the character 'U' (easily identifiable) and then overwrite the last byte (overflow) with 0x00 as explained above.

3. Integer Overflow

Vulnerability: Want to exploit `count = (long)strtoul(input, &in, 10);` which recasts an unsigned long to a signed long. This allows us to pass in an integer value that will be within the bounds of an unsigned long but overflow the signed long it is cast to and present therefore present as a negative value during `memcpy()` bounds check.

Exploit: Having passed the `memcpy()` bounds check, we can perform a similar exploit to q1 where we overwrite the return address of `foo` with the address of our shellcode, then execute the shellcode to get a root shell.

Working notes:

- Using gdb we can find address of the buffer variable, 0x7fffffe8f48. We then can find the address of the return address, 0x7fffffeed10.
- The difference of these two addresses is 24,000, which is the number of bytes we need to overwrite to get to the return address. Note, we need to add 8 bytes to this to account for the saved return address.
- To solve for count string to pass at the top of our exploit, we need to meet three conditions: (a) overflow the signed long, (b) be within the bounds of an unsigned long, and (c) solve for x such that $24x = 24024 \pmod{2^{64}}$. The first two conditions guarantee we pass the bounds check. The third condition ensures our exploit is large enough to overwrite the return address.
- We used a solver to find x such that $24x = 24024 \pmod{2^{64}}$. This gives us $x = 2,305,843,009,213,693,952n + 1,001$. We then solve for n such that $2^{63} < x < 2^{64}$ and find n can be 4, 5, 6, 7 or 8. We set $n = 4$ and use the resulting value and trailing comma ("9223372036854776809,") as our count string.
- Copy count string followed by our shellcode (without terminating null pointer) into our buffer exploit, and perform string surgery to ensure return address points back to beginning of our overwritten local buf variable.