CIS 751 Lecture Assignment 4

Chuck Zumbaugh

September 27, 2025

Let's assume we have two chunks of memory, A and B, where B has been freed and we can overflow A (into B). Since B is freed, it has two pointers, forward and backward, which will be used during the unlink procedure when free(A) is called. We can overwrite both of these pointers as we can overflow A into B. The unlink process is executed as:

$$*(\text{FWD\_pointer} + 12) = \text{BK\_pointer}$$
$$*(\text{BK\_pointer} + 8) = \text{FWD\_pointer}$$

We can use this to write whatever address is in BKpointer to FWDpointer + 12 (or vice versa), allowing us to write the address of malicious code somewhere in memory. Since library functions are stubbed with an address to the GOT, if we overwrite the GOT entry (-12) for a library function with the address to malicious code, the computer will begin executing our code when that function is called.

Assuming we can execute from the heap and the addresses are not randomized, we can craft a payload that writes the shell code in the data region of A, overwrites FWDpointer with the address to a GOT entry (some library function we know will be called), and overwrite BKpointer with the address of the start of the shell code. The heap will look something like the below figure.

Lower addresses
00000000
size a — 0/1
addr (begin shell code)
addr + 4
addr + 8
...
End shell code
00000000 (start of B metadata)
size b — 1
FWD pointer GOT address - 12
BK pointer addr
Rest of heap
Higher addresses

It is important to note that addr + 8 will be overwritten by the GOT address (when the second line of unlink is executed). Assuming our executable won't fit in the first 8 bytes, we will need to add a *jmp* instruction (5 bytes) to bypass it.

Alternatively, if the heap addresses are randomized but if we still have access to library addresses we can point the GOT entry at the system() function. However, we still need a way to give system it's arguments through the stack. When a function is called, the system will first push the arguments, push the EIP, and jump to the address of the call. Thus, system() will look for it's argument at EBP + 4. Although we don't control the stack, we can simulate this by pointing the overwritten GOT entry at a sequence of instructions that do the following:

```
; Set the address to a bin/sh string at what will be ebp + 4
push <addr to bin/sh>
push 0x41414141 ; junk address to return to after system returns
mov esp, ebp ; Set the base pointer to the top of the stack
jmp <addr of system()>
```

This assumes we can find a sequence of bytes in static libraries that allows us to do this. To do this, we would need to convert the above instructions to hex and perform a search in a known region of memory (for example in libc).