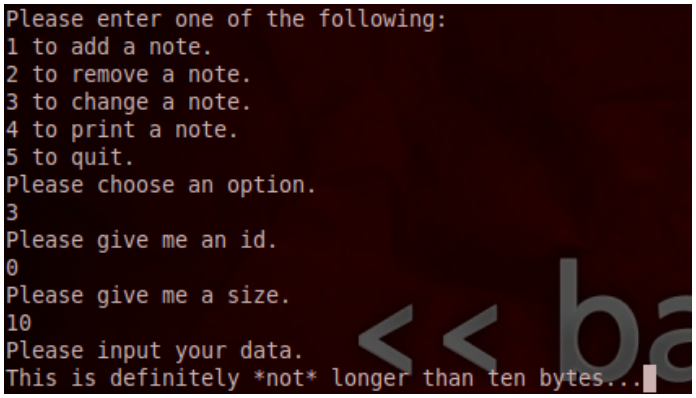
Team '; drop table *; --;-), PlaidCTF 2014, challenge “ezhp”

Given was the following task together with a linux executable:

> Luckily when you travel back in time, you still get to use all your knowledge from the present. With that knowledge in hand, breaking into this service (at 54.81.149.239:9174) owned by The Plague shouldn't be hard at all.

The service turns out to be a simple note manager - you can add notes, edit, read and delete them. What catches our attention is that, before entering your data for a note, the service asks for the length of the message - a dead giveaway on where to start analyzing.

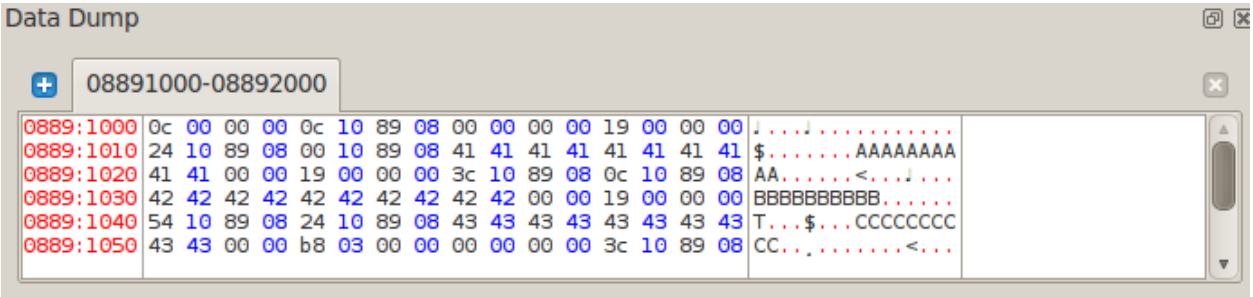


Using IDA, we track down the function responsible for reading our input.

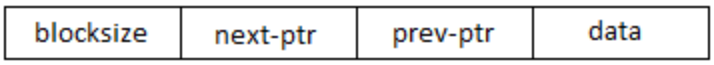
```
mov     eax, offset aDC ; "%d%*c"
lea     edx, [ebp+var_C]
mov     [esp+28h+var_24], edx
mov     [esp+28h+param_SIZE], eax
call    ___isoc99_scanf
mov     [esp+28h+param_SIZE], offset aPleaseInputYou ; "Please input your da
call    _puts
mov     eax, ds:stdout
mov     [esp+28h+param_SIZE], eax
call    _fflush
mov     edx, [ebp+var_C]
mov     eax, [ebp+id]
mov     eax, ds:dword_804A060[eax*4]
mov     [esp+28h+var_20], edx
mov     [esp+28h+var_24], eax
mov     [esp+28h+param_SIZE], 0
call    _read
jmp     short locret_8048954
```

```
; ||||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||

; Attributes: thunk

; int _read(int fildes,void *buf,size_t nbyte)
_read           proc near               ; CODE XREF: id_dialog_2+
                jmp     ds:off_804A000
_read           endp
```

The value we added in the request for the size of our note is clearly taken at face value and not compared to the actual size provided during the creation of the note. The new size is then provided as a parameter in the `read()` function, resulting in an overflow. To get a better understanding as to where our data is going to, we add a few notes and break at the `read()` function:



It turns out our data is transferred to the heap, into simplistic heap blocks. The structure of the blocks seems to be as follows:



Additionally, there is no randomization noticeable: if we repeat the process, the blocks always get ordered into memory in the same way. Due to the nature of double-linked lists, once an element `elem` is deleted, the following code sequence gets executed:

```
elem->next->prev = elem->prev;
elem->prev->next = elem->next;
```

Since we can overflow the data part of any block, we can overwrite both the next and the prev pointers in the blocks following the overflown block. With this knowledge, we can finally formulate our exploit:

First, we create three blocks in row. From the first block, we now overflow the complete second block, as well as the blocksize and the next-ptr of the third block. The next-ptr is overwritten by the address of the GOT-Entry for the exit()-function -8.

After that, we delete the third block. What happens next is this:

```
elem->next->prev = *((exit-8)+8) = *exit := prev-ptr
elem->prev->next := exit-8
```

The exit() function now points into our second block - which we can overwrite with our shellcode by overflowing the first block again. We overflow the buffer with a shellcode to start meterpreter, trigger the exploit by selecting an invalid option in the menu, and gain remote access with metasploit.

You can view the complete exploit on pastebin.

The flag is located in a file in the user ezhp's home directory: shitty_heap_allocators_are_shitty