

# Post Memory Corruption Memory Analysis



Jonathan Brossard  
CEO – Toucan System

jonathan@  
toucan-system.com



# Who am I ?

- Security Research Engineer at Toucan System
- Speaker at Blackhat, Defcon, HITB, H2HC, Kiwicon, Ruxcon.
- Organiser of the Hackito Ergo Sum conference (Paris).
- I'm the guy who comes to CCC with 90+ slides...

# I don't reverse plain text

I don't SQL inject

I don't PHP include

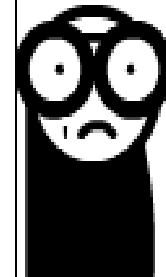
I don't Coldfusion

I don't .NET

I don't XSRF



\*sigh\*



# Agenda

---



**A few basics**



**Being environment aware**



**PMCMA Design**



**Extending Pmcma**



**•Stack desynchronization**



**Tool available at <http://www.pmcma.org>**

**We got 10k downloads+ in 2  
less than months...**

**... and every email we get is questioning  
exploitation of remote stack overflows  
instead of invalid memory writes... ;(**

# What's pmcma ?

It's a debugger, for Linux (maybe one day \*NIX) ptrace() based.

Pmcma allows to find and test **exploitation scenarios**.

Pmcma's output is a roadmap to exploitation, not exploit code.

Tells you if a given bug triggering an **invalid memory access** is a vulnerability, if it is exploitable with the state of the art, and how to exploit it.

What's pmcma ?

**DEMO**



**Coz you asked for it...**

# Remote stack overflow automated exploitation

**NX/SSP (stack cookies)/ASLR/PIE/STATIC  
GOT/Ascii Armoring...**

**=> No problem, easy cheesy : can be done  
with static analysis (of the libc/binary)  
only.**

# Remote stack overflow automated exploitation

**SSP :**

**cookies can be bruteforced remotely  
(cf Ben Hawks @ Ruxcon 2006).**

# Remote stack overflow automated exploitation

## FORTIFY :

- Doesn't apply all the time.
- Fails silently (this is bad !!)
- Is consistent under Linux (but not Apple...)

# Remote stack overflow automated exploitation

## PIE :

- The bug new thing (every deamon compiled with PIE under ubuntu 10.10)
- No public exploits (untill today ;)
- We can bruteforce the saved EIP, then get back to ret2plt or ROP.

**DEMO**

**Now, let's move to the real  
thing...**

# **A FEW BASICS**



**Seriously, can we skip this  
section ?**

# How do applications crash ?

- \* Stack corruptions -> stack overflows, usually now detected because of SSP | studied a LOT
- \* Signal 6 -> assert(), abort(): unexpected execution paths (assert() in particular), heap corruptions
- \* Segfault (Signal 11) -> Invalid memory access

# How do applications crash ?

- \* Stack corruptions -> stack overflows, usually now detected because of SSP | studied a LOT
- \* Signal 6 -> assert(), abort(): unexpected execution paths (assert() in particular), heap corruptions
- \* Segfault (Signal 11) -> Invalid memory access

# Invalid memory access

- trying to **read** a page not readable. often not mapped at all.
- trying to **write** to a page not writable. often not mapped at all.
- trying to **execute** a page not executable. often not mapped at all.

# Why do they happen ?

Because of any kind of miscomputation, really :

- integer overflows in loop counters or destination registers when copying/initializing data, casting errors when extending registers or
- uninitialised memory, dangling pointers
- variable misuse
- heap overflows (when inadvertently overwriting a function ptr)
- missing format strings
- overflows in heap, .data, .bss, or any other writable section (including shared libraries).
- stack overflows when no stack cookies are present...

# Exploiting invalid exec

Trivial, really. Eg :

```
call eax
```

with eax fully user controled

# Invalid memory reads (1/2)

Eg :

CVE-2011-0761 (Perl)

```
cmp BYTE PTR [ebx+0x8],0x9
```

# Invalid memory reads (2/2)

Eg :

CVE-2011-0764 (t1lib)

```
f1d  QWORD PTR [eax+0x8]
```



# Exploiting invalid memory reads ?

- usually plain not exploitable
- won't allow us to modify the memory of the mapping directly
- in theory : we could perform a user controlled read, to trigger a second (better) bug.

# Invalid memory writes

Eg :

CVE-2011-1824 (Opera)

```
mov    DWORD PTR [ebx+edx*1],eax
```

# How to...

To exploit invalid writes, we need to find ways to transform an arbitrary write into an arbitrary exec.

The most obvious targets are function pointers.

# Exploiting invalid memory writes : scenario

- Target a known function pointer (typically : `.dtors`, GOT entry...).

Can be prevented at compile time : no `.dtors`, static GOT...

- Target function pointers in the whole binary ?
- Overwrite a given location to trigger an other bug (eg : stack overflow)

**Being environment aware**

# Problems to take into account

- Kernel : ASLR ? NX ?
- Compilation/linking : RELRO  
(partial/full) ? no .dtors section ? SSP ?  
FORTIFY\_SOURCE ?

=> Pmcma needs to measure/detect those features

# ASLR

**Major problem when choosing an exploitation strategy.**

# ASLR : not perfect

- Prelinking (default on Fedora) breaks ASLR
- All kernels don't have the same randomization strength.
- Non PIE binaries

=> Truth is : we need better tools to test it !



# Testing ASLR

- Run a binary  $X$  times (say  $X=100$ )
  - Stop execution after loading
- Record mappings.

=> Compare mappings, deduce randomization

**DEMO : being environment aware**

# **PMICMA DESIGN**

# GOALS

- We want to test overwriting different memory locations inside a process and see if they have an influence over the flow of execution
- We want to scale to big applications (web browsers, network deamons...)
- We want a decent execution time

# mk\_fork()

The idea :

- We start analysing after a SEGFAULT
- We make the process fork() (many many times)
- Inside each offspring, we overwrite a different memory location

# `mk_fork()` : benefits

Mapping looks « just like » it will when actually exploiting a binary

No ASLR/mapping replication problem

Exhaustive and hopefully fast

# How to force a process to fork ?

- 1) Find a +X location mapped in memory.
- 2) Save registers
- 3) Use ptrace() to inject fork() shellcode.
- 4) Modify registers so eip points to shellcode.
- 5) Execute shellcode.
- 6) Wait() for both original process and offspring.
- 7) Restore bytes in both processes.
- 8) Restore registers in both processes.

# Forking shellcode

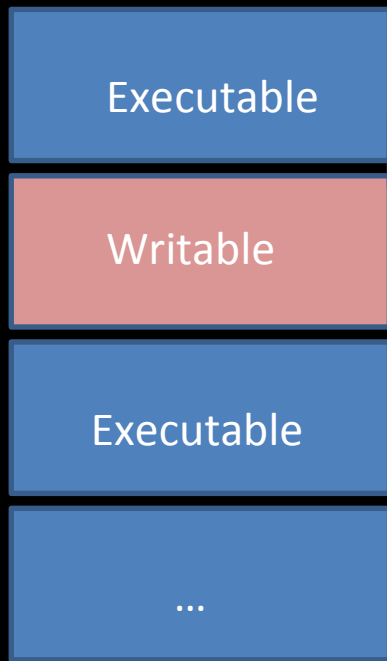
**;forking shellcode:**

00000000	6631C0	xor eax,eax
00000003	B002	mov al,0x2
00000005	CD80	int 0x80

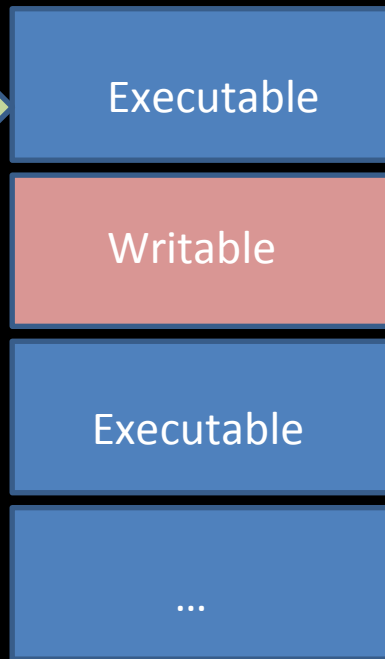


# mk\_fork()

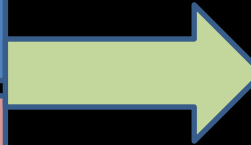
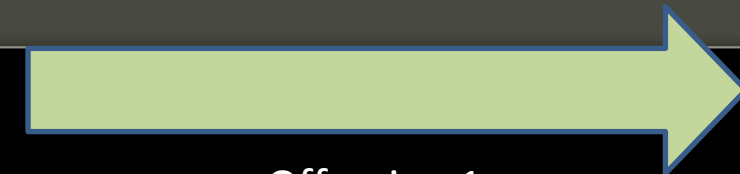
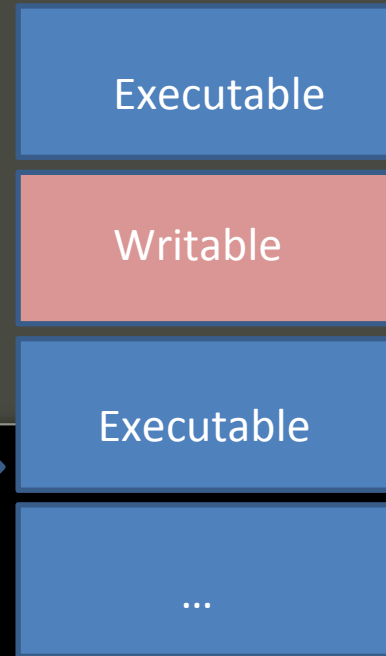
Original process



Offspring 1



Offspring 2



# mk\_fork()

Offspring 1

Executable

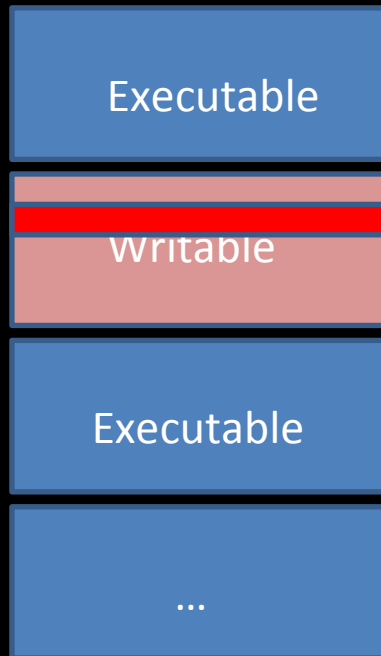
Writable

Executable

...

# mk\_fork()

Offspring 2



# mk\_fork()

Offspring n

Executable

Writable

Executable

...

# `mk_fork()` : PROS

- allows for multiple tests out of a single process
- fast, efficient (no recording of memory snapshots)
- no need to use breakpoints
- no single stepping

# `mk_fork()` : CONS

- Dealing with offsprings termination ?  
(Zombie processes)
- I/O, IPC, network sockets will be in unpredictable state
- Hence syscalls will get wrong too (!!)

# Zombie reaping

- Avoid the `wait()` for a `SIGCHLD` in the parent process.
- Kill processes after a given timeout, including all of their children.

# Zombie reaping : the SIGCHLD problem

If we can have the parent process ignore  
SIGCHLD signals, we won't create  
Zombies.

=> We inject a small shellcode to perform  
this via `sigaction()`



# Zombie reaping : the SIGCHLD problem

- 1) Find a +X location mapped in memory.
- 2) Save registers
- 3) Use ptrace() to inject sigaction() shellcode.
- 4) Modify registers so eip points to shellcode.
- 5) Execute shellcode.
- 6) Wait() for the process while executing shellcode.
- 7) Restore bytes in +X location.
- 8) Restore registers in the process.



**Zombie reaping : killing the  
offsprings and their children**

**Fortunately, this is possible using « process  
grouping »...**

# Process grouping

`setpgid()` sets the PGID of the process specified by `pid` to `pgid`. If `pid` is zero, then the process ID of the calling process is used. If `pgid` is zero, then the PGID of the process specified by `pid` is made the same as its process ID. If `setpgid()` is used to move a process from one process group to another (as is done by some shells when creating pipelines), both process groups must be part of the same session (see `setsid(2)` and `credentials(7)`). In this case, the `pgid` specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

# Zombie reaping : forcing process grouping

- 1) Find a +X location mapped in memory.
- 2) Save registers
- 3) Use ptrace() to inject setpgid() shellcode.
- 4) Modify registers so eip points to shellcode.
- 5) Execute shellcode.
- 6) Wait() for the process while executing shellcode.
- 7) Restore bytes in +X location.
- 8) Restore registers in the process.

# Force process grouping...

```
;  
; setpgid(0,0); shellcode  
;
```

```
_start:  
  nop  
  nop  
  nop  
  nop  
  mov eax,0x39    ; setpgid  
  xor ebx,ebx  
  xor ecx,ecx  
  int 0x80  
  
  db 0xcc, 0xcc
```

# Zombie reaping : final details

From now on, to kill a process and all of its children :

`kill (-pid, SIGTERM) ;`

# IPC, I/O, invalid syscalls

One possibility is to recode correct execution on the original process (after clearing signals and ignoring the **SEGFault**).

Then replay/fake the syscalls on the offsprings.

=> Minimal userland « virtualization ».



# **PMICMA : FEATURES**

# Exploiting invalid memory writes via function pointers

We now want to find all the function pointers called by the application from the instruction which triggered the `SEGFAULT` until it actually halts.

(including pointers in shared libraries!!)

# Finding all the function pointers actually called

- 1) Parse all the +W memory, look for possible pointers to any section
  - 1 bis) optionally disassemble the destination and see if it is a proper prologue.
- 2) use `mk_fork()` to create many children
- 3) in each children, overwrite a different possible function pointer with a canari value (0xf1f2f3f4).
- 4) Monitor execution of the offsprings

# Finding all the function pointers actually called

Overwritten pointer leads to execution of canari address 0xf1f2f3f4

<=> We found a called function pointer.

Finding all the function  
pointers actually called

DEMO

So what can we test now ?

**Invalid write anything anywhere :**

attacker has full control over data written  
and destination where written

**=> GAME OVER**

So what can we test now ?

**Overflows** (in any writable section but the stack) :

Simply limit the results of pmcma to this section.

So what can we test now ?

What if the attacker has little or **no control over the data being written** (arbitrary write non controlled data, anywhere) ?



# Partial overwrites and pointers truncation

If we can't properly overwrite a function pointer, maybe we can still **truncate** one (with the data we don't control) so that it transfers execution to a controlled memory zone ?

# Exemple :

--[ Function pointers exploitable by truncation with 0x41424344:

At 0xb70ce070 : 0xb70c63c2 will become 0xb70c4142 (lower truncated by 16 bits, dest perms:RW)

At 0xb70e40a4 : 0xb70ca8f2 will become 0xb70c4142 (lower truncated by 16 bits, dest perms:RW)

At 0xb70ec080 : 0xb70e5e02 will become 0xb70e4142 (lower truncated by 16 bits, dest perms:RW)

At 0xb731a030 : 0xb7315da2 will become 0xb7314142 (lower truncated by 16 bits, dest perms:RW)

At 0xb73230a4 : 0xb732003a will become 0xb7324142 (lower truncated by 16 bits, dest perms:RW)

At 0xb732803c : 0xb7325a36 will become 0xb7324142 (lower truncated by 16 bits, dest perms:RW)

At 0xb76a80d8 : 0xb7325bf0 will become 0xb7324142 (lower truncated by 16 bits, dest perms:RW)

# One more situation...

**Sometimes, an attacker has limited control over the destination of the write (whether he controls the data being written or not).**

**Eg : 4b aligned memory writes.**

# Exploiting 4b aligned memory writes

We can't attack a function pointer directly,  
unless it is unaligned (rare because of  
compiler internals).

Pmcma will still let you know if this  
happens ;)

# Exploiting 4b aligned memory writes : plan B

Find all « normal » variables we can overwrite/truncate, and attempt to trigger a second bug because of this overwrite.

# Finding all unaligned memory accesses

Setting the **unaligned flag** in the EFLAGS register will trigger a signal 7 upon next access of unaligned memory (read/write).

# Finding all unaligned memory accesses

DEMO

# Finding all unaligned memory accesses

DEMO x86\_64



# Defeating ASLR : Automated memory mapping leakage

How does WTFuzz did it at CansecWest  
2010 to win the pwn2own contest against  
IE8/Windows 7 ?

Overwrite the null terminator of a JS  
string to perform a mem leak uppon  
usage (trailing bytes).

# Defeating ASLR with an arbitrary write ?

In the original process :

- use `ptrace() PTRACE_SYSCALL`
- record the calls to `sys_write()` and `sys_socketall()` (wrapper to `sys_send()` or `sys_sendto()`...), including : where is the data sent ? How many bytes ?

# Defeating ASLR with an arbitrary write ?

Create many offsprings using `mk_fork()`.

-In each of them : overwrite a different location with dummy data.

-Follow execution using `PTRACE_SYSCALL`

-Monitor differences : a different address or a bigger size means a memory leak :)

# Extending Pmcma

# Means of modifying the flow of execution without function pointers

Call tables.

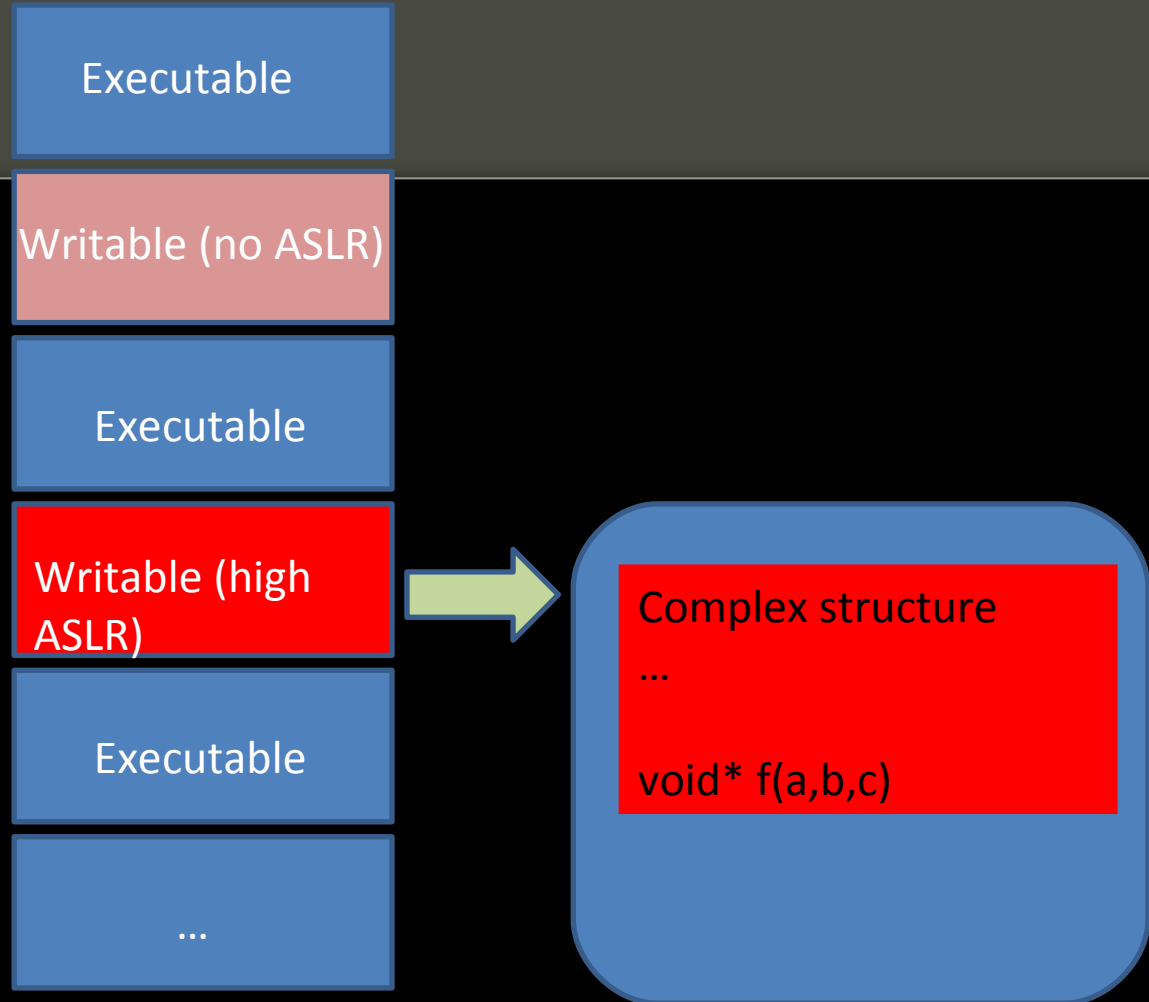
Calling [Offset+register]

=> This is also already performed  
automatically using pmcma.

# Pointers and ASLR

If overwriting a given function pointer isn't practical because of ASLR : is it possible to overwrite a pointer (in an other section) to a structure containing this function pointer ? Would this « other section » be less randomised ?

# Finding pointers to structures containing function pointers



# Finding pointers to structures containing function pointers

We'd like to have the debugged process create a new section, with a given mapping (to ease identify).

Modify a possible pointer per offspring (use `mk_fork()`).

Monitor execution : is the offspring calling a function pointer from our custom mapping ?



# Forcing a process to create a new mapping :

- 1) Find a +X location mapped in memory.
- 2) Save registers
- 3) Use ptrace() to inject mmap() shellcode.
- 4) Modify registers so eip points to shellcode.
- 5) Execute shellcode.
- 6) Wait() for the process while executing shellcode.
- 7) Restore bytes in +X location.
- 8) Restore registers in the process.

```
;
; old_mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0) shellcode:
;
```

```
_start:
```

```
    nop
    nop
    nop
    nop
```

```
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx
    xor esi, esi
    xor edi, edi
```

```
    mov bx, 0x1000          ; 1 page
    mov cl, 0x3            ; PROT_READ | PROT_WRITE
    mov dl, 0x21          ; MAP_SHARED | MAP_ANON
```

```
    push eax
    push eax
    push edx
    push ecx
    push ebx
    push eax
```

```
    mov ebx, esp
    mov al, 0x5a          ; sys_mmap
    int 0x80
```

```
; eax contains address of new mapping
```

```
    db 0xcc, 0xcc, 0xcc, 0xcc
```

# Testing exhaustively arbitrary writes

In case all of the above failed...

Can we trigger secondary bugs by overwriting specific memory locations ?

# Testing exhaustively arbitrary writes

Complexity is huge !

Still doable with Pmcma, with no guaranty  
over the time of execution.

# Testing exhaustively arbitrary reads

In the same veine, attacker controled invalid reads can trigger secondary bugs, which will be exploitable.

=> We can test the whole 4+ billions search space (under x86 Intel architecture), or just a few evenly chosen ones.

# Stack desynchronization

W^X is a problem.

Even if we can overwrite fully a function pointer and modify the flow of execution...  
what do we want to execute in 2011 ?

# Stack desynchronization

Instead of returning directly to shellcode in +W section (hence probably not +X) :

- Return to a function epilogue chosen so that esp will be set to user controlled data in the stack.
- Fake stack frames in the stack itself.
- Use your favorite ROP/ret2plt shellcode

# Stack desynchronization :

## Exemple : sudo

- stack is ~1000 big (at analysis time)
- we find a function pointer to overwrite (at 0x0806700c)
- we overwrite it with a carefully chosen prologue (inc esp by more than 1000)



# Stack desynchronization :

## Exemple : sudo

```
jonathan@blackbox:~$ objdump -Intel -d /usr/bin/sudo
```

```
...  
805277a: 81 c4 20 20 00 00  add esp,0x2020  
8052780: 5b                pop  ebx  
8052781: 5e                pop  esi  
8052782: 5d                pop  ebp  
8052783: c3                ret
```

# Stack desynchronization : Exemple : sudo

We can control the destination where esp is going to point : simply use an environment variable

```
TOTO=mydata sudo
```

# Stack desynchronization :

## Exemple : sudo

We then forge fake stack frames in the stack itself

- « Nop sled » : any pointer to 'ret'

Eg :804997b: c3 ret

- Then copy shellcode to .bss byte per byte using memcpy via ret2plt

- Use GOT overwrite to get pointer to mprotect() in the GOT (ROP)

- call mprotect to make .bss +X via ret2plt

- return to shellcode in .bss



**DEMOS**

# Future Work

- port to more architectures (Linux x86\_64 on the way, arm...)
- port to more OS (Mac OSX, \*BSD)
- port to Windows (hard)
- add tests for other bug classes

# Thank you for coming

## Questions ?

