

# Assembleur et X86 (cdecl) calling convention

Nicolas Gama

# Reverse engineering

- 3 types:
  - Analyse statique
    - Lecture et compréhension du code assembleur (souvent avec ida pro)
  - Debogage dynamique
    - Execution assembleur ligne à ligne
    - Visualisation/modification des variables et registres
  - Patch persistant
    - Modifier le code machine, par ex: no-cd cracks

# GDB mode assembleur

```
bash$ gdb ./programme

# mettre un point d'arrêt sur un nom de fonction
b main

# Ou si on a une adresse d'instruction
b * 0x48010230

# lancer l'execution (jusqu'au prochain point d'arrêt)
r param1 param2

# voir le code assembleur de la fonction courante
disas

# executer l'instruction assembleur et s'arreter à la suivante
ni

# continuer jusqu'au prochain breakpoint
c

# afficher un registre, par ex %eax ou %esp
p $eax
P $esp #ici, c'est sans doute une adresse

# changer la valeur de %eax
set $eax=0
```

# GDB mode assembleur

```
# Dumper le contenu d'une zone mémoire  
# (et formater l'affichage)
```

```
x/15xb 0xffffffff230
```

```
0xffffffff230: 0x70 0x07 0x40 0x00 0x00 0x00 0x00 0x00  
0xffffffff238: 0x40 0xfa 0xa2 0xf7 0xff 0x7f 0x00
```

```
x/4xw 0xffffffff230
```

```
0xffffffff230: 0x00400770 0x00000000 0xf7a2fa40 0x00007fff
```

```
# n nombre d'éléments à afficher
```

```
# format:
```

```
# x hexadécimal
```

```
# s null-terminated string
```

```
# i machine instruction
```

```
# ou encore: d, f, u comme dans printf
```

```
# size: (pas pour s évidemment !)
```

```
# b 1 byte
```

```
# h 2 bytes
```

```
# w 4 bytes
```

```
# g 8 bytes
```

# Ok d'accord...

- IDA et GDB ont l'air pas mal...
- Mais concrètement, ça marche comment l'assembleur?
- Et surtout, quelle variable dois-je afficher pour résoudre les challenges???

# C++ versus Assembleur

- Langage haut niveau (c++, java, C...)
  - Notions de fonctions (arguments, retour)
  - Notions de variables locales/globales
- Assembleur
  - Une mémoire: random access table, indexée par des adresses, et qui contient
    - Des données (valeurs ou pointeurs)
    - Du code machine (payloads)
  - Des registres (valeurs ou pointeurs)

# Langages haut niveau

- Un pseudo-code qui traduit des concepts mathématiques et algorithmiques
- Des boucles: for, while
- Des appels récursifs

# Assembleur

- Des instructions arithmétiques simples
- Des variables globales:
  - Registres (eax, ecx, edx, eip, ebp, ...)
  - Le tableau de mémoire
- Des pointeurs
  - Si une variable contient une adresse, on peut accéder au contenu de la mémoire à cette adresse
- Des sauts (GOTO), éventuellement conditionnels



# Compilation

- Le compilateur traduit un langage haut niveau au langage assembleur (lisible).
- Le code assembleur est en bijection explicite avec le code machine (binaire)
- Il existe plein de manières de compiler un code, pour être générique, le compilateur suit des conventions de traduction (convention d'appels, calling convention). Par ex:
  - cdecl (32 bits, X86)
  - x86\_64

# Ce que la machine impose

- 1 registre très particulier:
  - **\$eip**: contient l'adresse de l'instruction en cours d'exécution courante.
    - eip est en général incrémenté après chaque instruction
    - Mais en cas de saut, le programme est libre de fixer le prochain eip.
      - Instruction jb: saut inconditionnel
      - Instruction jz, jnz...: saut conditionnel
  - Beaucoup d'attaques tentent de contrôler ce registre!

# La pile (stack)

- C'est une zone vers le haut du tableau de mémoire, qui grandit vers le bas.
- Le registre **\$esp** contient l'adresse du sommet de la pile
- Fait la liaison entre la vue haut niveau et l'assembleur bas niveau.

# Instructions manipulant la pile

- **Pushl** x: ajoute l'entier x au sommet de la pile
  - Pseudo-code équivalent:
    - $[\$esp] := x$
    - $\$esp := \$esp - 4$
- **Push** \$eax: ajoute le contenu du registre \$eax au sommet de la pile
- **Pop** \$eax: dépile le sommet dans \$eax

# Call et Ret

- **Call** addr:
  - Pseudo-code équivalent:
    - push \$eip+1
    - \$eip := addr
- **Ret:**
  - Pseudo-code équivalent:
    - Pop \$eip
  - *Cela ne fonctionnera “normalement” que si l'ancien \$eip est bien au sommet de la pile!!*

# Example

code	stack
-----	-----
%eip => 0x00001f66: call 0x1ef0 <nop_ret>	0xdeadbeef   <= %esp
0x00001f6b: [...]	

code	stack
-----	-----
%eip => 0x00001ef0: ret	0xdeadbeef
	0x00001f6b   <= %esp

code	stack
-----	-----
0x00001f66: call 0x1ef0 <nop_ret>	0xdeadbeef   <= %esp
%eip => 0x00001f6b: [...]	

# Arguments

- Dans la convention cdecl, les arguments des fonctions sont placés sur la pile juste avant le call.
- (Le premier argument est empilé en dernier: il reste au sommet)
- Cette convention concerne quasiment toutes les fonctions C, sauf main().

# Exemple

Appeler une fonction proj\_1(0x05,0x10)

	code		stack
%eip =>	0x00001f78: pushl \$0x10		0xdeadbeef   <= %esp
	0x00001f7a: pushl \$0x5		
	0x00001f7c: call 0x1f90 <proj_1>		
	0x00001f81: addl \$0x8, %esp		



# Exemple

Appeler une fonction proj\_1(0x05,0x10)

code				stack	
-----				-----	
%eip =>	0x00001f78:	pushl	\$0x10		
	0x00001f7a:	pushl	\$0x5		
	0x00001f7c:	call	0x1f90 <proj_1>		
	0x00001f81:	addl	\$0x8, %esp		

# Exemple

Appeler une fonction proj\_1(0x05,0x10)

code				stack	
-----				-----	
	0x00001f78:	pushl	\$0x10	0xdeadbeef	
	0x00001f7a:	pushl	\$0x5	0x00000010	
%eip =>	0x00001f7c:	call	0x1f90 <proj_1>	0x00000005	<= %esp
	0x00001f81:	addl	\$0x8, %esp		

**Remarque:** comme la pile grandit vers le bas, les adresses sont au final dans le même ordre que dans la fonction!

# Exemple (bis)

Appeler une fonction proj\_1(0x05,0x10)

**Remarque2:** Ce code alternatif est plus courant, et donnera exactement le même effet!

	code		stack
%eip =>	0x00001f78: subl \$0x8, \$esp		0xdeadbeef <= %esp
	0x00001f7a: movl \$0x10, 4(\$esp)		
	0x00001f7c: movl \$0x5, (\$esp)		
	0x00001f7e: call 0x1f90 <proj_1>		
	0x00001f81: addl \$0x8, %esp		

# Exemple (bis)

Appeler une fonction proj\_1(0x05,0x10)

**Remarque2:** Ce code alternatif est plus courant, et donnera exactement le même effet!

code				stack	
-----				-----	
%eip =>	0x00001f78:	subl	\$0x8, \$esp		0xdeadbeef
	0x00001f7a:	movl	\$0x10, 4(\$esp)		
	0x00001f7c:	movl	\$0x5, (\$esp)		
	0x00001f7e:	call	0x1f90 <proj_1>		
	0x00001f81:	addl	\$0x8, %esp		
					<= %esp

# Exemple (bis)

Appeler une fonction proj\_1(0x05,0x10)

**Remarque2:** Ce code alternatif est plus courant, et donnera exactement le même effet!

	code		stack	
	-----	+	-----	
	0x00001f78: subl	\$0x8, \$esp	0xdeadbeef	
	0x00001f7a: movl	\$0x10, 4(\$esp)	0x00000010	
%eip =>	0x00001f7c: movl	\$0x5, (\$esp)		<= %esp
	0x00001f7e: call	0x1f90 <proj_1>		
	0x00001f81: addl	\$0x8, %esp		

# Exemple (bis)

Appeler une fonction proj\_1(0x05,0x10)

**Remarque2:** Ce code alternatif est plus courant, et donnera exactement le même effet!

code			stack	
-----			-----	
%eip =>	0x00001f78:	subl \$0x8, \$esp	0xdeadbeef	<= %esp
	0x00001f7a:	movl \$0x10, 4(\$esp)	0x00000010	
	0x00001f7c:	movl \$0x5, (\$esp)	0x00000005	
	0x00001f7e:	call 0x1f90 <proj_1>		
	0x00001f81:	addl \$0x8, %esp		

# Exemple (bis)

Appeler une fonction proj\_1(0x05,0x10)

**Remarque3:** A la fin de l'appel, c'est l'appelant qui fait le ménage dans la pile

code			stack	
-----			-----	
0x00001f78:	subl	\$0x8, \$esp	0xdeadbeef	<= %esp
0x00001f7a:	movl	\$0x10, 4(\$esp)	0x00000010	
0x00001f7c:	movl	\$0x5, (\$esp)	0x00000005	
0x00001f7e:	call	0x1f90 <proj_1>		
%eip => 0x00001f81:	addl	\$0x8, %esp		

# Exemple (bis)

Appeler une fonction proj\_1(0x05,0x10)

**Remarque3:** A la fin de l'appel, c'est l'appelant qui fait le ménage dans la pile

*(L'appelant doit juste remettre \$esp en place  
Il n'est pas nécessaire de remettre à 0 le contenu)*

code			stack	
-----			-----	
0x00001f78:	subl	\$0x8, \$esp	0xdeadbeef	<= %esp
0x00001f7a:	movl	\$0x10, 4(\$esp)	0x00000010	
0x00001f7c:	movl	\$0x5, (\$esp)	0x00000005	
0x00001f7e:	call	0x1f90 <proj_1>		
0x00001f81:	addl	\$0x8, %esp		
%eip => [...]				



# Base pointer

- En plus des arguments ou des adresses de retour, les fonctions peuvent utiliser la pile pour stocker des variables locales.
- Le “base pointer” \$ebp sert par convention à indiquer le début de la zone de la pile qui correspond à la fonction courante.
- Pour que cela fonctionne, la première instruction de chaque fonction est copie l'ancien \$ebp sur la pile

# Code utilisant \$ebp

```
my_function:
    push %ebp                # Preamble: save the old %ebp.
    movl %esp, %ebp          # Point %ebp to the saved %ebp and the new stack frame.

    subl $0x8, %esp          # Reserve space for two (4 bytes) local variables.

    movl 0x8(%ebp), %eax      # copy second argument...
    movl %eax, -0x4(%ebp)     # ... into the first local variable.

    # Function body.

    addl $0x8, %esp           # Reclaim space used by local variables.

    pop %ebp                 # Epilogue: restore the old %ebp.
    ret
```

# Contenu de la pile au milieu de la fonction

<argument 2>	
<argument 1>	
<return address>	
<old ebp>	<= %ebp
<local var 1>	
<local var 2>	<= %esp

## Remarque:

La gestion de \$ebp est entièrement manuelle, et c'est uniquement une histoire de convention d'appels.

Ce n'est pas comme \$eip et \$esp qui “bougent” automatiquement avec les instructions adéquates (call, ret, push, pop)

# Registres, et return value

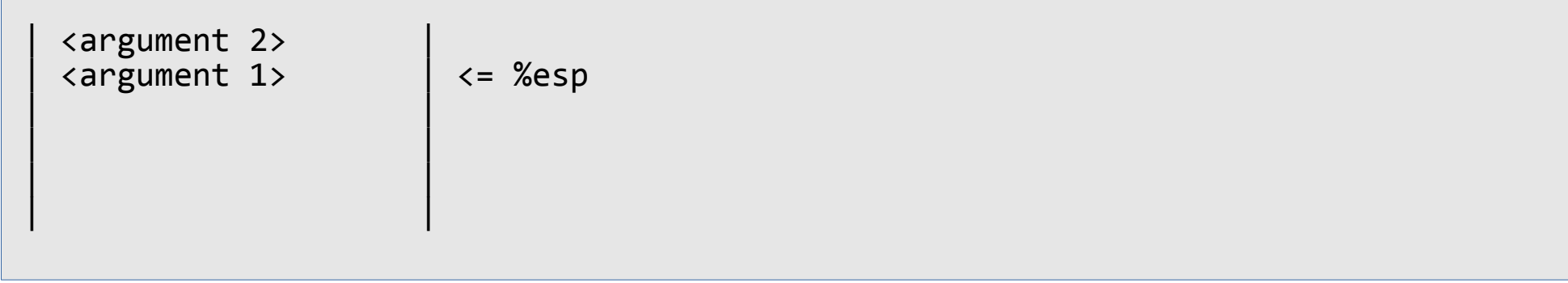
- Un programme peut utiliser librement des registres \$eax, \$ecx, \$edx, ...
- La valeur des registres n'est pas garantie de persister d'un appel à l'autre
  - Mais on peut toujours les sauver dans des variables locales (sur la pile)
- Si elle existe, la valeur de retour doit être placée dans \$eax lors du ret.
  - Les structures sont retournées par pointeur.

# En Résumé

- **\$esp** pointe vers le dernier élément empilé (sommet du stack)
- **\$eip** pointe vers la prochaine instruction
- **Call addr**: push \$eip et change \$eip := addr
- **Ret**: pop le sommet dans \$eip

# En Résumé

- Les arguments sont empilés avant le call
- **Juste avant** le call, la pile ressemble à cà:



The diagram illustrates a stack frame. It consists of a light gray rectangular area. On the left side, there are two vertical lines. Between these lines, the text "<argument 2>" is positioned above "<argument 1>". To the right of the second vertical line, the text "<= %esp" is displayed.

```
<argument 2>  
<argument 1>
```

```
<= %esp
```

# En Résumé

- Juste **après** le call, la pile ressemble à ça:  
(à l'entrée de la fonction)

<code>&lt;argument 2&gt;</code>	<code>&lt;= %esp</code>
<code>&lt;argument 1&gt;</code>	
<code>&lt;return address&gt;</code>	

# En Résumé

- Au milieu de la fonction (après la “déclaration” des variables locales et mise à jour de \$ebp):

<argument 2>	
<argument 1>	
<return address>	
<old ebp>	<= %ebp
<local var 1>	
<local var 2>	<= %esp



# En Résumé

- A la fin de la fonction (après nettoyage des variables locales):
- La valeur de retour est dans \$eax
- Et la pile redevient:

<code>&lt;argument 2&gt;</code>	<code>&lt;= %esp</code>
<code>&lt;argument 1&gt;</code>	
<code>&lt;return address&gt;</code>	

# En Résumé

- Après le **ret**
- *C'est à l'appelant de dépiler les arguments!*

<code>&lt;argument 2&gt;</code> <code>&lt;argument 1&gt;</code>	<code>&lt;= %esp</code>
--	-------------------------

# Différences 32/64 bits

- Les registres s'appellent %rip, %rsp, %rax, ...
  - Les versions %eip, %esp... existent, mais sont tronquées à 32 bits.
- Les 6 premiers arguments entiers sont passés dans les registres %rdi, %rsi, %rdx, %rcx, %r9, %r8, seulement les autres sont empilés.
  - Les 8 premiers flottants sont passés dans les registres %xmm0 à %xmm7.