

CYBV 471 Assembly Programming for Security Professionals Week 10

Stack, function, and assembly instructions

Agenda



- **➤** What is the Stack?
- **Calling Function Procedure**
- **➤** Assembly Language Instructions
 - ➤ Understand "MOV" instruction
 - ➤ Understand "NOP" instruction
 - ➤ Understand "LEA" instruction
 - ➤ Understand "NOP" instruction
 - ➤ Understand "LEA" instruction
 - ➤ Understand "PUSH" instruction
 - ➤ Understand "POP" instruction
 - ➤ Understand "CALL" instruction
 - ➤ Understand "RET" instruction

Mov vs LEA Instructions



- mov destination, source
 - Moves (copies) data from one location to another (RAM/Register)
- Operand surrounded by brackets references to data located in memory location
 - [ebx] points to data located in the memory address stored in EBX
 - [0x4037c4] points to data located in memory address 0x4037c4

Instruction	Description
mov eax, ebx	Copies the contents of EBX into the EAX register
mov eax, 0x42	Copies the value 0x42 into the EAX register
mov eax, [0x4037C4]	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
mov eax, [ebx]	Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
mov eax, [ebx+esi*4]	Copies the 4 bytes at the memory location specified by the result of the equation ebx+esi*4 into the EAX register

Load Effective Address (LEA) instruction



- lea destination, source
 - Put a memory address (located in source) into the destination
- lea eax, [ebx+8]
 - Puts memory address (ebx+8) into eax
- In contrast

```
mov eax, [ebx+8]
```

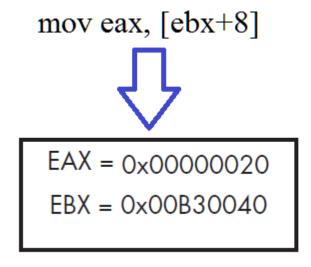
Moves the data located in memory address pointed by (ebx+8) into eax

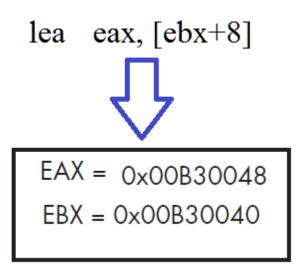
mov & lea Instruction Examples



 Values for registers EAX and EBX on the left and the information contained in memory on the right







NOP instruction



- Does nothing, proceed to the next instruction
- The opcode for this instruction is 0x90
- Used to pad/align bytes, or to delay time

What is a Stack?



- A stack is a temporary memory storage region that holds a function variables, data, registers values.
- Every function has its stack frame

Main Memory

Stack

Local variables and parameters for functions Helps programs flow

Heap

Dynamic memory

Changes frequently during program execution Program allocates new values, and frees them when they are no longer needed

Code

Instructions for the CPU

Controls what the program does

Data

Static values placed when a program loads

They cannot change while the program is running

Global values available to any part of the program

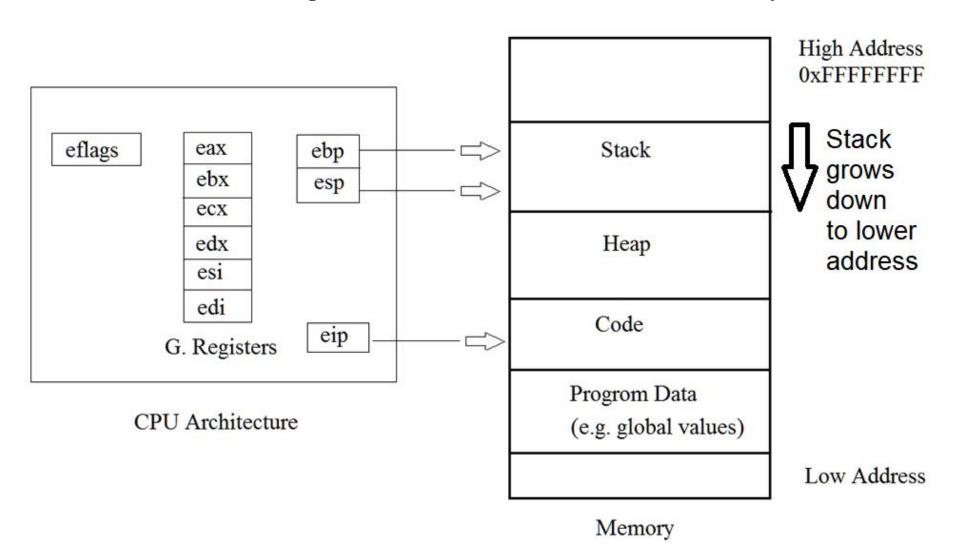
n	High
n-1	Memory
n-2	Address

2 Low1 Memory0 Address

What is a Stack?



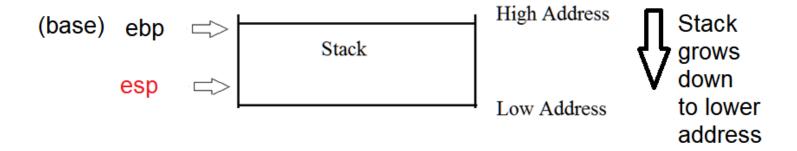
- The stack grows down towards lower addresses
- The value ESP is the "top" of the stack or the lowest address used by the stack

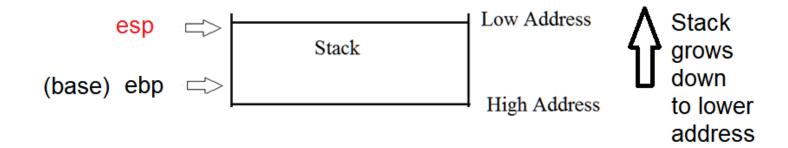


Visualizing the stack



- Some books show stack grows down towards low addresses
- Some books show stack grows up towards low addresses
- Debuggers show stack grows up towards low addresses





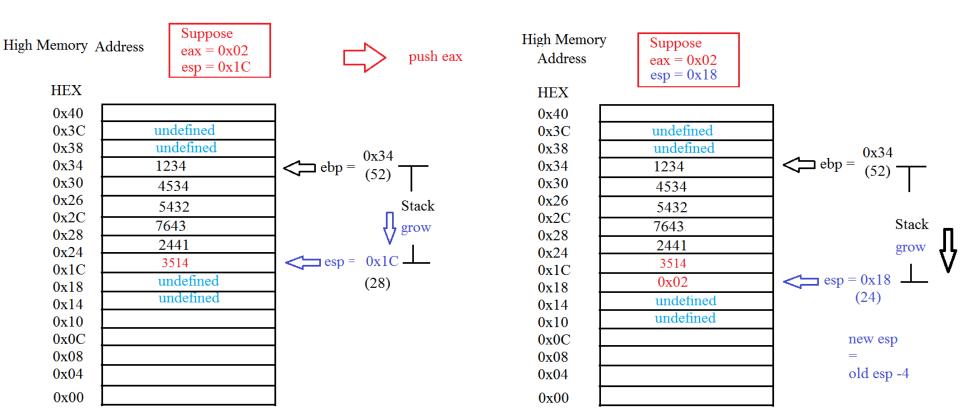
PUSH Instruction



- Pushes a DWORD onto the stack
 - can either be an immediate (a numeric constant), or the value in a register
- Two operations in one (in order)
 - 1. Adjusts the stack pointer (New ESP = Old ESP 4)
 - 2. Writes the new address value to ESP
- Note that if a register is pushed, the value in the register will not change

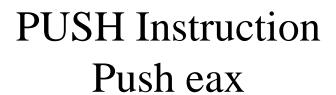
PUSH Instruction



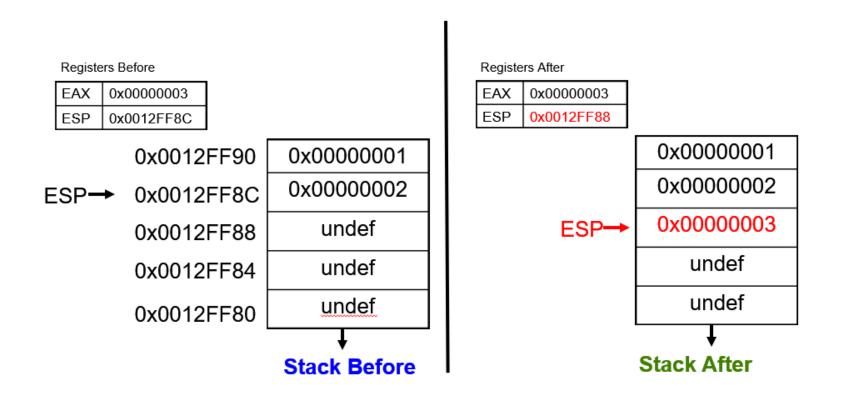


Low Memory Address

push: add value to the stack esp: decresed by 4







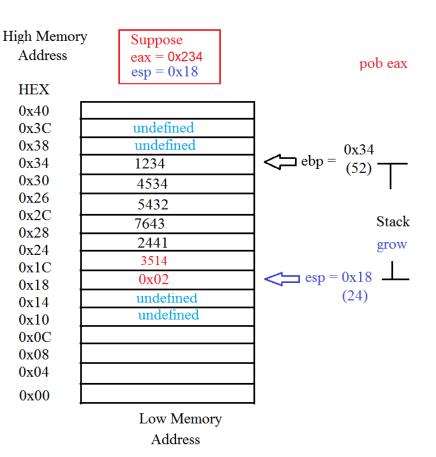
POP Instruction

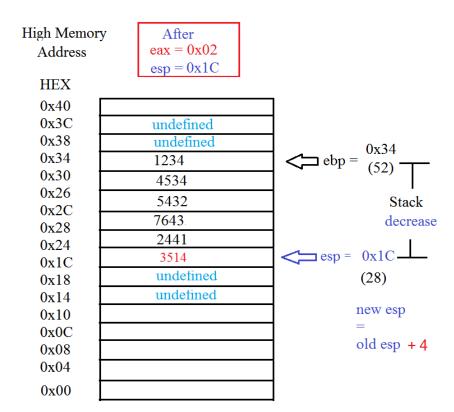


- POPs a DWORD from the top of the stack INTO a register
- Two operations in one (in order)
 - 1. Adjusts the stack pointer (New ESP = Old ESP + 4)
 - 2. Writes the new address value to ESP
- Note that the value on the stack is not changed; only ESP changes

POP Instruction



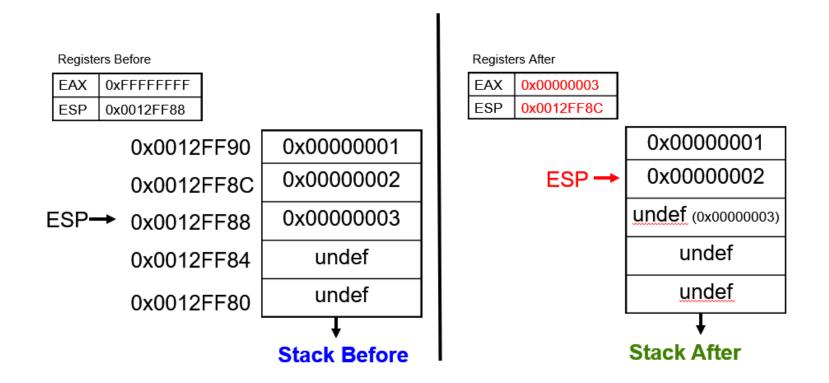




pob: take off the top value of the stake esp: increased by 4 (moves to higher memory addresss)

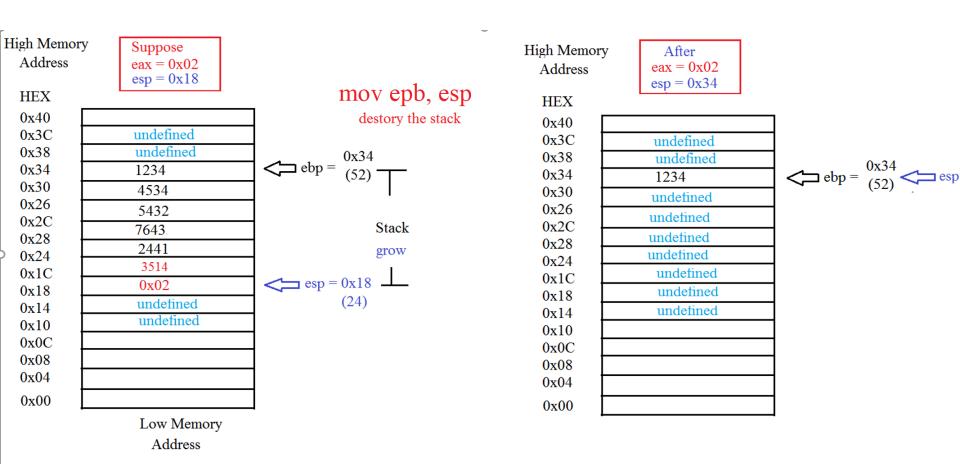


POP Instruction POP eax



Destroy a Function stack





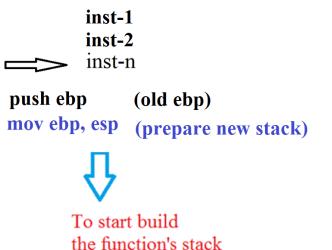
Function Calls

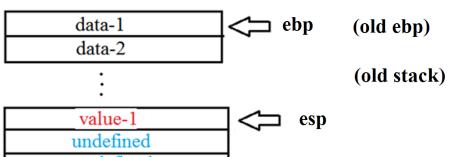


- Functions are portions of code within a program that perform a specific task
- Functions are relatively independent of the remaining code.
- The main code calls and temporarily transfers execution to functions before returning to the main code.
- Each time a call is performed, a new stack frame is generated
 Prologue: Instructions at the start of a function that prepare stack and registers for the function to use
- A function maintains its own stack frame until it returns.
 Epilogue: Instructions at the end of a end of a function that restore the stack and registers to their state before the function was called

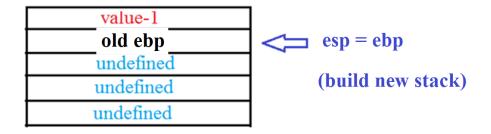








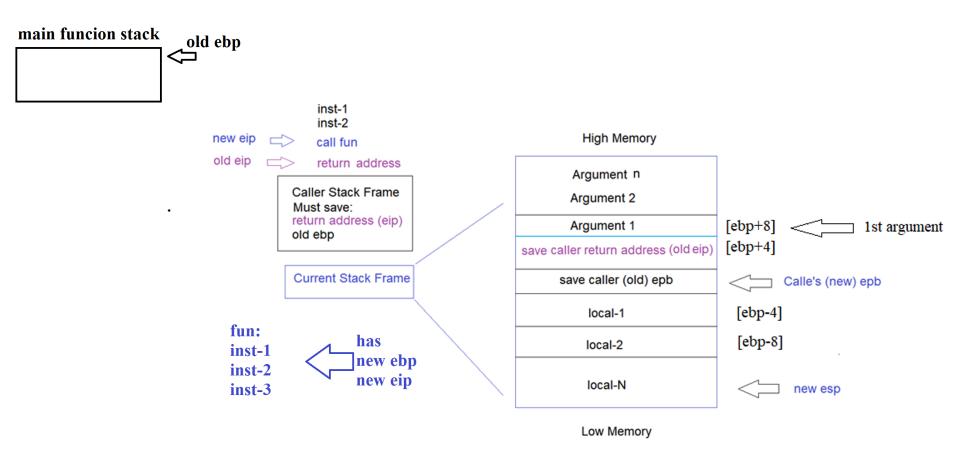
Start build a function stack



CALL – Call Procedure



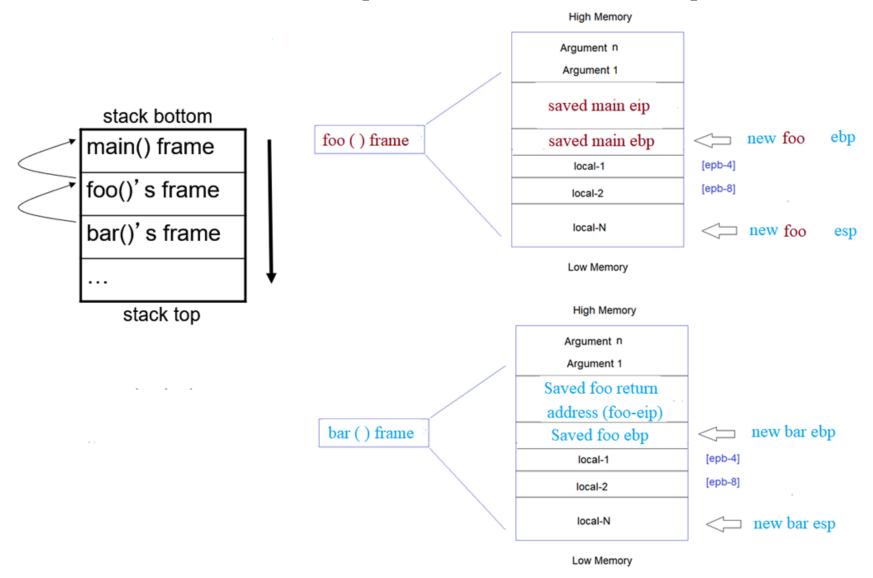
- Two operations in one (in order):
 - 1. PUSH the address of the next instruction (old eip value) onto the stack
 - 2. Set new EIP to the CALL's destination address



Stack Frames are a Linked List



The EBP in the current frame points to the saved EBP of the previous frame.



Function epilogue



- Free the called function variables
- Restore the old pointer (old eip) of the calling function
- Return the control to the calling function

mov ebp, esp ; destroy the stack
pob ebp ; get old ebp value
ret ; pob old eip ; get old eip value

RET – Return from Procedure



- Ret: The function returns by calling the ret instruction. This pops the return address off the stack into EIP (get the old EIP value), so that the program will continue executing from where the original call was made.
- If "RET <number>", two steps:
 - a- POPs the top of the stack into EIP as before
 - b- Adds < number > to EAX
 - Examples: RET 0x8 or RET 0x20
- Leave: Sets ESP to equal EBP (destroy the stack) and pops EBP off the stack (get the old EBP value)



Registers and calling conventions

- Stack conventions:
 - −ESP − Top of the stack
 - -EBP Pointer to last stack frame (next slide)
- Register saving *conventions*
 - EBX, ESI, EDI and EBP persist across CALLs
 - Callers MUST ASSUME that EAX, ECX, EDX and ESP will change
 - Callers must save its register values to avoid any issues program crash





Example 1.c



The stack frames in this example will be very simple.

Only saved frame pointer (EBP) and saved return addresses (EIP)

```
sub:
                         00401000 PUSH EBP
int sub()
                         00401001 MOV ESP, EBP
 return Oxbeef;
                         00401003 MOV EAX, 0BEEFh
                         00401008 POP EBP
                         00401009 RET
int main()
                         main:
                         00401010 PUSH EBP
 sub();
                         00401011 MOV ESP, EBP
 return 0xf00d;
                         00401013 CALL SUB (401000h)
                         00401018 MOV EAX,0F00Dh
                         0040101D POP EBP
                         0040101E RET
```



Assume: EIP = 00401010, but no instruction yet executed

eax	0x003435C0 ₩
ebp	0x0012FFB8 ₩
esp	0x0012FF6C ₩



EX executed instruction,

M modified value

¥ start value

sub:

00401000 push ebp

00401001 mov ebp,esp

00401003 mov eax,0BEEFh

00401008 pop ebp

00401009 ret

>> main: (start point)

00401010 push ebp

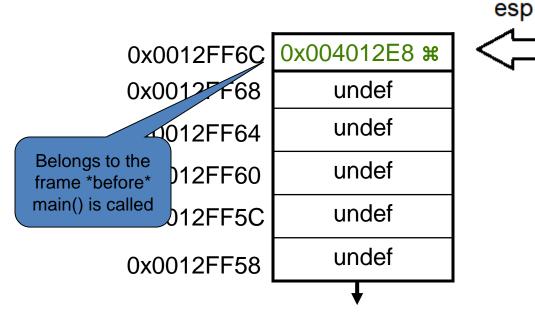
00401011 mov ebp,esp

00401013 call sub (401000h)

00401018 mov eax,0F00Dh

0040101D pop ebp

0040101E ret





Before

00401000 push

eax	0x003435C0 ₩
ebp	0x0012FFB8 ₩
esp	0x0012FF6C ₩

Key:

x executed instruction,

modified value

x start value

eax	0x003435C0
ebp	0x0012FFB8
esp	0x0012FF68 M)

After

sub:

	-	-
00401001	mov	ebp,esp
00401003	mov	eax,0BEEFh
00401008	pop	ebp
00401009	ret	
main:		
00401010	push	ebp 🗵
00401011	mov	ebp, esp
00401013	call	sub (401000h)
00401018	mov	eax,0F00Dh
0040101D	pop	ebp
0040101E	ret	

ebp

)x0012FF6C	0x004012E8 %
)x0012FF68	0x0012FFB8 m
)x0012FF64	undef
)x0012FF60	undef
)x0012FF5C	undef
)x0012FF58	undef
•	





esp

eax	0x003435C0 ₩
ebp	0x0012FFB8 ₩
esp	0x0012FF68 %

Key:

区 executed instruction,

modified value

¥ start value

eax	0x003435C0 業
ebp	0x0012FF68 11)
esp	0x0012FF68

sub:

0040101E ret

00401000 push	ebp	ebp	0x0012FF6C	0x004012E8 \$
00401001 mov	ebp,esp	\Box	0x0012FF68	0x0012FFB8
00401003 mov	eax,0BEEFh		0x0012FF64	undef
00401008 pop 00401009 ret	ebp		0x0012FF60	undef
main:			0x0012FF5C	undef
00401010 push	ebp			undef
00401011 mov	ebp, esp 🗵 (ebp	= esp)	0x0012FF58	T T T T T T T T T T T T T T T T T T T
00401013 call	sub (401000h)			•
00401018 mov	eax,0F00Dh			
0040101D pop	ebp			



eax	0x003435C0 ₩
ebp	0x0012FF68
esp	0x0012FF68

区 executed instruction,

modified value

x start value

eax	0x003435C0 ₩
ebp	0x0012FF68
esp	0x0012FF64 M

sub:

0040101E ret

00401000	push	ebp		ebp
00401001	mov	ebp,esp		<u>,</u>
00401003	mov	eax,0BI	EEFh	<u> </u>
00401008	pop	ebp	Save return	addre
00401009	ret		(old eip)	
main:				
00401010	push	ebp		
00401011	mov	ebp,esp		
00401013	call	sub (401	1000h) 🗵	
00401018	mov	eax,0F0	0Dh	
0040101D	pop	ebp		

0x0012FF6C	0x004012E8 x
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	old eip
0x0012FF5C	undef
	undef
0x0012FF58	u n def





eax	0x003435C0 ₩
ebp	0x0012FF68 ₩
esp	0x0012FF64 %

Key	
-----	--

EX executed instruction,

modified value

x start value

eax	0x003435C0 ₩
ebp	0x0012FF68
esp	0x0012FF60 M

sub:

00401000

0040101E ret

00401000	push	ebp 🗵	ebp
00401001	mov	ebp,esp	
00401003	mov	eax,0BEEFh	<u> </u>
00401008	pop	ebp	
00401009	ret	Save call	er (old) ebp
main:			
00401010	push	ebp	
00401011	mov	ebp,esp	
00401013	call	sub (401000h)	
00401018	mov	eax,0F00Dh	
0040101D	pop	ebp	

0x0012FF6C 0x0012FF68 0x0012FF64 ebp 0x0012FF60 0x0012FF5C 0x0012FF58

0x004012E8 **%**0x0012FFB8
0x00401018
old eip
0x0012FF68 m
undef
undef



eax	0x003435C0 ₩
ebp	0x0012FF68
esp	0x0012FF60 ₩

0040101D pop

0040101E ret

ebp



executed instruction, modified value

x start value

eax	0x003435C0 ₩
ebp	0x0012FF60 M
esp	0x0012FF60

sub:		eip = 00401003			
00401000 push	ebp	Build new stack		0.00404050	
00401001 mov	ebp,esp 🗵	frame for sub ()	0x0012FF6C	0x004012E8 %	
00401003 mov	eax,0BEEFh	Traine for suc ()	0x0012FF68	0x0012FFB8	
00401008 pop	ebp	ebp	0x0012FF64	0x00401018	esp
00401009 ret		N	0.00405500	0x0012FF68	/Sp
main:		>	0x0012FF60	00012660	egthinspace = egt
00401010 push	ebp		0x0012FF5C	undef	,
00401011 mov	ebp,esp		0.00405550	undef	
00401013 call	sub (401000h)		0x0012FF58	dilaci	
00401018 mov	eax,0F00Dh			\	



eax	0x003435C0 ₩
ebp	0x0012FF60 %
esp	0x0012FF60 ₩

cuh.

Kev	٠
	c

x executed instruction,

start value

eax	0x0000BEEF 10
ebp	0x0012FF60
esp	0x0012FF60

Sub:					
00401000 push	ebp	eip= 00401008			1
00401001 mov	ebp,esp	сір оонотооо	0x0012FF6C	0x004012E8 	
00401003 mov	eax,0BEEFh 🗵	3	0x0012FF68	0x0012FFB8	
00401008 pop	ebp		0,00121100		
00401009 ret	·	ebp	0x0012FF64	0x00401018	esp
main:		_/	0.00405500	0x0012FF68	
00401010 push	ebp	└ √	0x0012FF60	UXUUTZFF00	7
00401011 mov	ebp,esp		0x0012FF5C	undef	
00401013 call	sub (401000h)		0,00121100		
00401018 mov	eax,0F00Dh		0x0012FF58	undef	
0040101D pop	ebp			1	
0040101E ret				▼	

eax	0x0000BEEF ≭
ebp	0x0012FF60 %
esp	0x0012FF60 ¥

Key:

X executed instruction,

modified value

start value

	4
eax	0x0000BEEF
ebp	0x0012FF68 M
esp	0x0012FF64 M

eip = 00401009

sub: 00401000 push ebp 00401001 mov ebp,esp eax,0BEEFh 00401003 mov 00401008 pop ebp 🗵 00401009 ret main:

00401010 push ebp 00401011 mov ebp,esp 00401013 call sub (401000h) 00401018 mov eax.0F00Dh 0040101D pop ebp

0040101E ret

Note that a POP doesn't actually modify the value, but it should be considered undefined by

Notice that ebp got old value saved before calling sub()

eip = 00401009**SF68** 0x0012FF64 0x0012FF60 0x0012FF5C 0x0012FF58

Q12FF6C

0x004012E8 ** 0x0012FFB8 0x00401018 undef m undef undef Notice

saved

old eip

(return address)

esp

eax	0x0000BEEF
ebp	0x0012FF68 Ж
esp	0x0012FF64 Ж

Key:

⊠ executed instruction,

nodified value

x start value

_	
eax	0x0000BEEF
ebp	0x0012FF68
esp	0x0012FF68 M

Notice that eip got old value saved before calling sub()

eip = 00401018

sub:

00401000 push ebp 00401001 mov ebp,esp

00401003 mov eax,0BEEFh

00401008 pop ebp

00401009 ret **⊠**

main:

00401010 push ebp

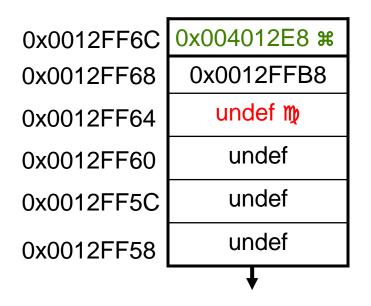
00401011 mov ebp,esp

00401013 call sub (401000h)

00401018 mov eax,0F00Dh

0040101D pop ebp

0040101E ret



esp



eax	0x0000BEEF ૠ
ebp	0x0012FF68
esp	0x0012FF68

Key:

EX executed instruction,

modified value

¥ start value

eax	0x0000F00D M
ebp	0x0012FF68
esp	0x0012FF68

eip = 0040101D

c	11	h	•	
3	u	U	•	

00401000 push ebp 00401001 mov ebp,esp 00401003 mov eax,0BEEFh

00401008 pop ebp

00401009 ret

main:

00401010 push ebp

00401011 mov ebp,esp

00401013 call sub (401000h) 00401018 mov eax,0F00Dh ☒

0040101D pop ebp

0040101E ret





0x0012FF6C 0x0012FF68

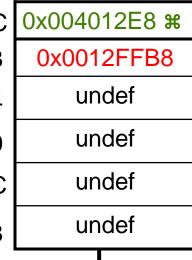
0000126600

0x0012FF64

0x0012FF60

0x0012FF5C

0x0012FF58





eax	0x0000F00D
ebp	0x0012FF68
esp	0x0012FF68 ૠ

Kov	
IXCY	•

☒ executed instruction,

modified value

% start value

	/A \
eax	0x0000Fบบบ
ebp	0x0012FFB8 M
esp	0x0012FF6C M

Notice that ebp got old value saved before calling main ()

CII	h	•
Su	v	•

00401000 push ebp

00401001 mov ebp,esp

00401003 mov eax,0BEEFh

00401008 pop ebp

00401009 ret

main:

00401010 push ebp

00401011 mov ebp,esp

00401013 call sub (401000h) 00401018 mov eax,0F00Dh

0040101D pop ebp ⊠

0040101E ret

0x0012FF68

0x0012FF64

0x0012FF60

0x0012FF5C

0x0012FF58

0x004012E8 %		
undef m		
undef		

eax	0x0000F00D
ebp	0x0012FFB8 ₩
esp	0x0012FF6C ₩

Key	
-----	--

x executed instruction,

modified value

第 start value

	/A \
eax	0x0000F00D
ebp	0x0012FFB8
esp	0x0012FF70 11)

sub:

00401000 push ebp 00401001 mov ebp,esp 00401003 mov eax,0BEEFh 00401008 pop ebp 00401009 ret main: 00401010 push ebp 00401011 mov ebp,esp 00401013 call sub (401000h) 00401018 mov eax,0F00Dh 0040101D pop ebp 0040101E ret **⊠**

0x0012FF70

0x0012FF6C	undef 🐚
0x0012FF68	undef
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef
•	

Execution would continue at the value ret removed from the stack: 0x004012E8

New eip = 0x004012E8

Putting It All Together



You should know:

- > What is the Stack?
- **Calling Function Procedure**
- > Assembly Language Instructions
 - ➤ Understand "MOV" instruction
 - ➤ Understand "PUSH" instruction
 - ➤ Understand "POP" instruction
 - ➤ Understand "CALL" instruction
 - ➤ Understand "RET" instruction
 - ➤ Understand "NOP" instruction
 - ➤ Understand "LEA" instruction



Questions?

Coming Next Week More Assembly Language Instructions

Week 10 Assignments



Learning Materials

- 1- Week 6 Presentation
- 2- Read Pages 246-251 (Duntermann, Jeff. Assembly Language Step by Step, Programming with Linux)
- 3- Read Pages 65-75 (PCASM textbook)

Assignment

1- Complete "Lab 10" by coming Sunday 11:59 PM.