INTRODUCTION TO ASSEMBLY X86

INTRODUCTION

- Avinash Kumar Thapa
- Working with EmiratesNBD as Senior InfoSec SME.
- OSCP and OSCE
- Bug hunter on Hackerone, Synack

CREDITS & REFERENCES

- All credit to opensecuritytraining.info

AGENDA

- The intent of this class is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.

WHAT ARE YOU GOING TO LEARN

```
#include <stdio.h>
int main(){
    printf(“Hello World!\n”);
    return 0x1234;
}
```

IS SAME AS...

```
.text:00401730 main
   .text:00401730                push    ebp
   .text:00401731                mov     ebp, esp
   .text:00401733                push    offset aHelloWorld ; "Hello
   world\n"
   .text:00401738                call    ds:__imp__printf
.text:0040173E                   add     esp, 4
.text:00401741                   mov     eax, 1234h
.text:00401746                   pop     ebp
.text:00401747                   retn
```

Windows Visual C++, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version

IS SAME AS..

Ubuntu Disassembled with “objdump -d”

```
08048374 <main>:
 8048374:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048378:       83 e4 f0                and    $0xfffffff0,%esp
 804837b:       ff 71 fc                 pushl  -0x4(%ecx)
 804837e:       55                       push   %ebp
 804837f:       89 e5                    mov    %esp,%ebp
 8048381:       51                       push   %ecx
 8048382:       83 ec 04                 sub    $0x4,%esp
 8048385:       c7 04 24 60 84 04 08      movl   $0x8048460,(%esp)
 804838c:       e8 43 ff ff ff           call   80482d4 <puts@plt>
 8048391:       b8 2a 00 00 00           mov    $0x1234,%eax
 8048396:       83 c4 04                 add    $0x4,%esp
 8048399:       59                       pop    %ecx
 804839a:       5d                       pop    %ebp
 804839b:       8d 61 fc                lea    -0x4(%ecx),%esp
 804839e:       c3                       ret
 804839f:       90                       nop
```

IS SAME AS ..!

Mac OS Disassembled from command line with “otool -tV”

```
_main:
00001fca  pushl  %ebp
00001fcb  movl  %esp,%ebp
00001fcd  pushl  %ebx
00001fce  subl  $0x14,%esp
00001fd1  calll  0x00001fd6
00001fd6  popl  %ebx
00001fd7  leal  0x0000001a(%ebx),%eax
00001fdd  movl  %eax,(%esp)
00001fe0  calll  0x00003005  ; symbol stub for: _puts
00001fe5  movl  $0x00001234,%eax
00001fea  addl  $0x14,%esp
00001fed  popl  %ebx
00001fee  leave
00001fef  ret
```

- By one measure, only 14 assembly instructions account for 90% of code!
- I think that knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently
- You've already seen 11 instructions, just in the hello world variations!

ARCHITECTURE – REGISTERS

- Registers are small memory storage areas built into the processor (still volatile memory)

- 8 “general purpose” registers + the instruction pointer which points at the next instruction to execute

- On x86-32, registers are 32 bits long

- On x86-64, they're 64 bits

ARCHITECTURE – REGISTERS CONVENTIONS -1

•These are Intel's suggestions to compiler developers (and assembly handcoders). Registers don't have to be used these ways, but if you see them being used like this, you'll know why. But I simplified some descriptions. I also colour coded as **GREEN** for the ones which we will actually see in *this* class (as opposed to future ones), and **RED** for not.

- **EAX** – Stores function return values
- **EBX** – Base pointer to the data section
- **ECX** – Counter for string and loop operations
- **EDX** – I/O pointer

ARCHITECTURE- REGISTERS – CONVENTION 2

- **ESI** – Source pointer for string operations
- **EDI** – Destination pointer for string operations
- **ESP** – Stack pointer
- **EBP** – Stack frame base pointer
- **EIP** - Pointer to next instruction to execute (“instruction pointer”)

ARCHITECTURE – REGISTERS – 8/16/32 BIT ADDRESSING

<table>
<tr><th colspan="32">8/16/32bit general purpose registers</th></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">EAX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">AX</td></tr>
<tr><td colspan="8">AH</td><td colspan="8">AL</td></tr>
<tr><td colspan="32">ECX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">CX</td></tr>
<tr><td colspan="8">CH</td><td colspan="8">CL</td></tr>
<tr><td colspan="32">EDX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">DX</td></tr>
<tr><td colspan="8">DH</td><td colspan="8">DL</td></tr>
<tr><td colspan="32">EBX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">BX</td></tr>
<tr><td colspan="8">BH</td><td colspan="8">BL</td></tr>
</table>

ARCHITECTURE – REGISTERS – 8/16/32 BIT ADDRESSING-2

16/32bit general purpose registers																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ESP																															
reserved																						SP									
EBP																															
reserved																						BP									
ESI																															
reserved																						SI									
EDI																															
reserved																						DI									

ARCHITECTURE- EFLAGS

•EFLAGS register holds many single bit flags. Will only ask you to remember the following for now.

–**Zero Flag (ZF)** - Set if the result of some instruction is zero; cleared otherwise.
–**Sign Flag (SF)** - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

YOUR FIRST INSTRUCTION – NOP

- NOP - No Operation! No registers, no values, no nothin'!

- Just there to pad/align bytes, or to delay time

- Bad guys use it to make simple exploits more reliable. But that's another class ;)

- "The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction."

PUSH INSTRUCTION – 2ND INSTRUCTION

- For our purposes, it will always be a DWORD (4 bytes).
  - Can either be an immediate (a numeric constant), or the value in a register
- The push instruction automatically decrements the stack pointer, esp, by 4.

Registers Before

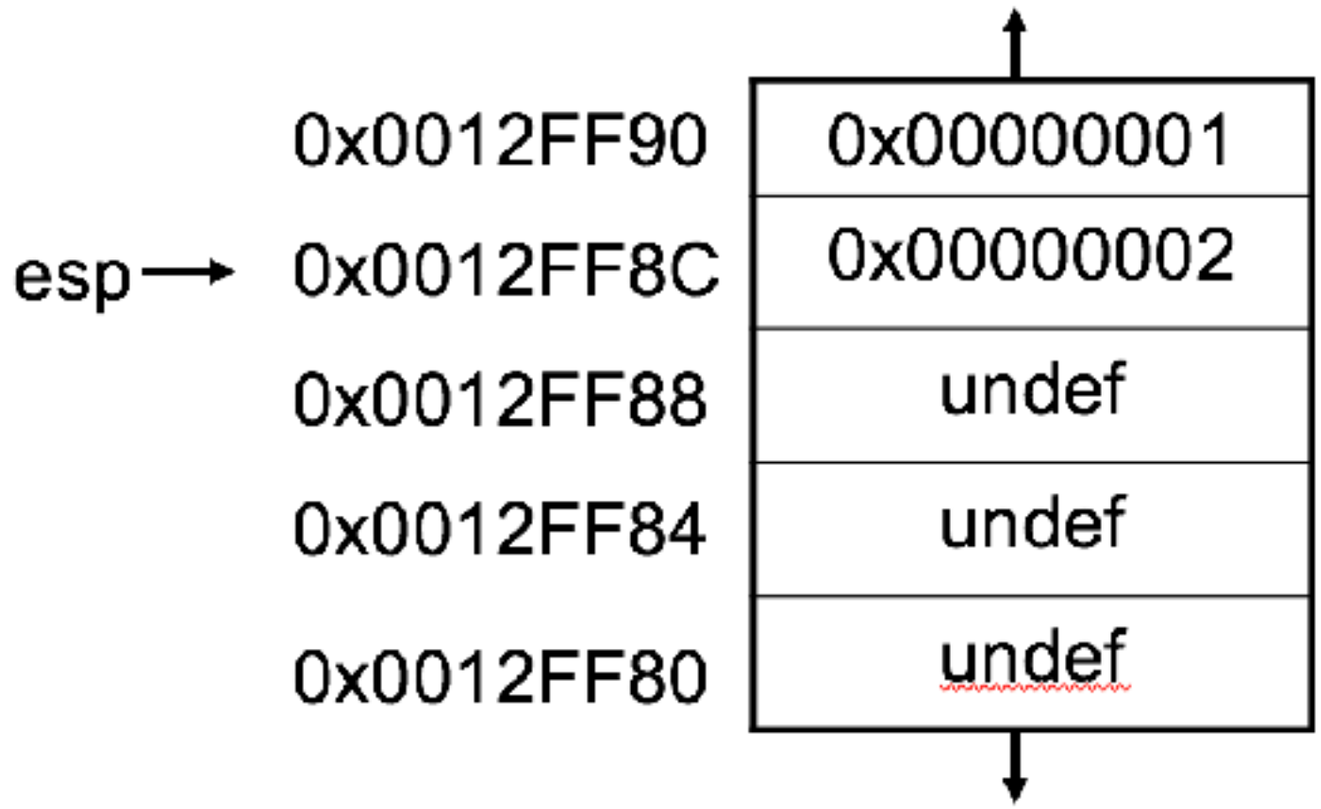
eax	0x00000003
esp	0x0012FF8C

push eax

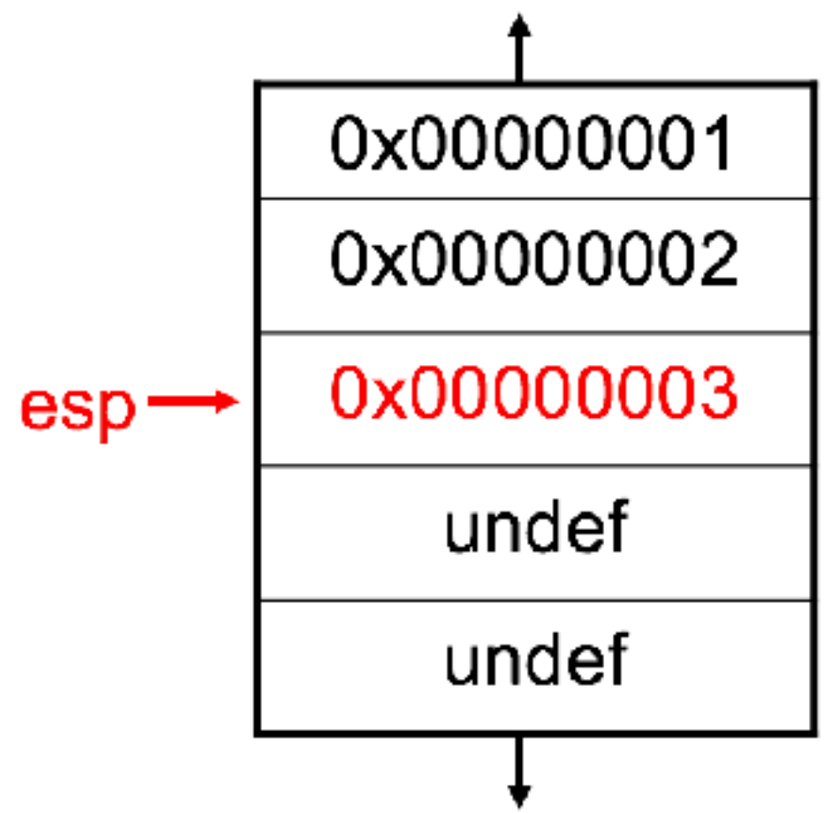
Registers After

eax	0x00000003
esp	0x0012FF88

Stack Before



Stack After



POP INSTRUCTION – 3RD

- Take a DWORD off the stack, put it in a register, and increment esp by 4

pop eax

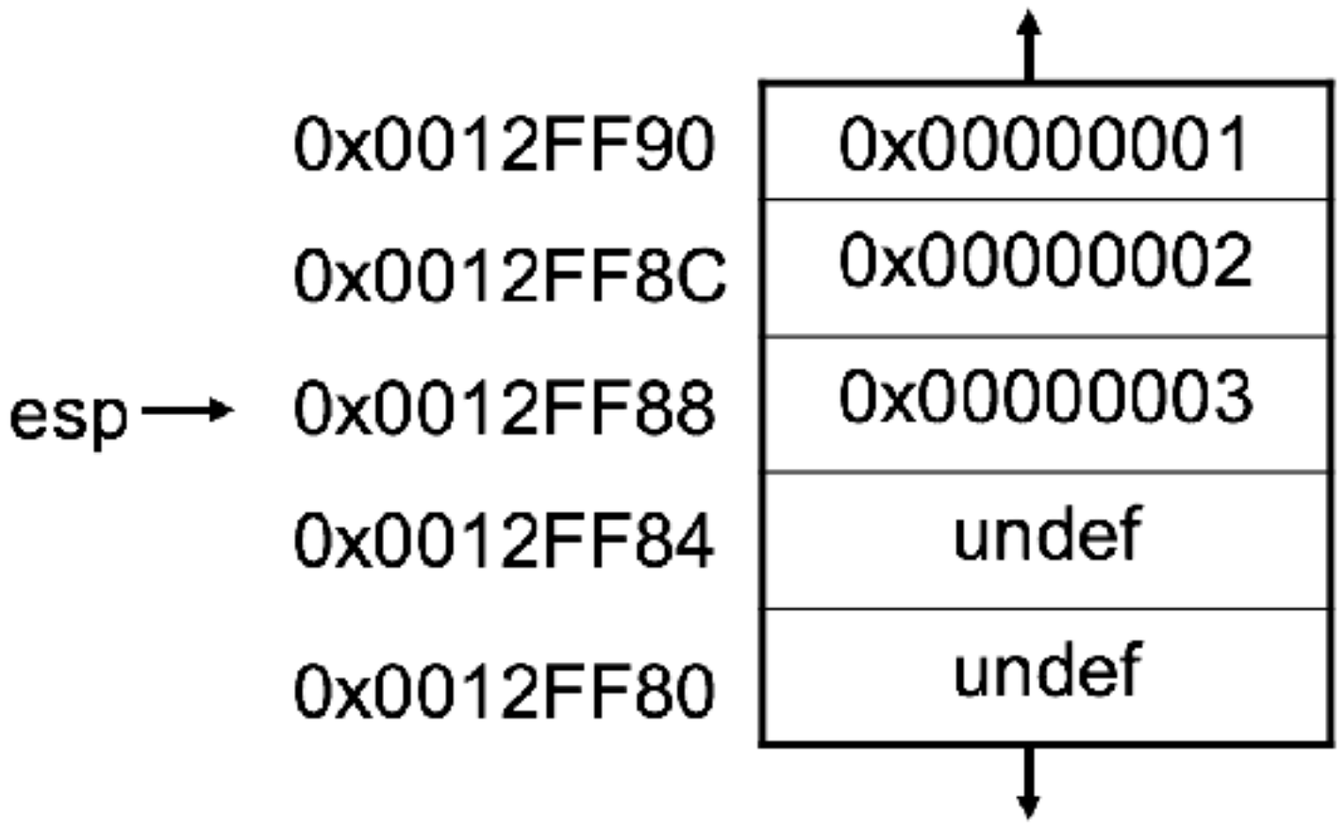
Registers Before

eax	0xFFFFFFFF
esp	0x0012FF88

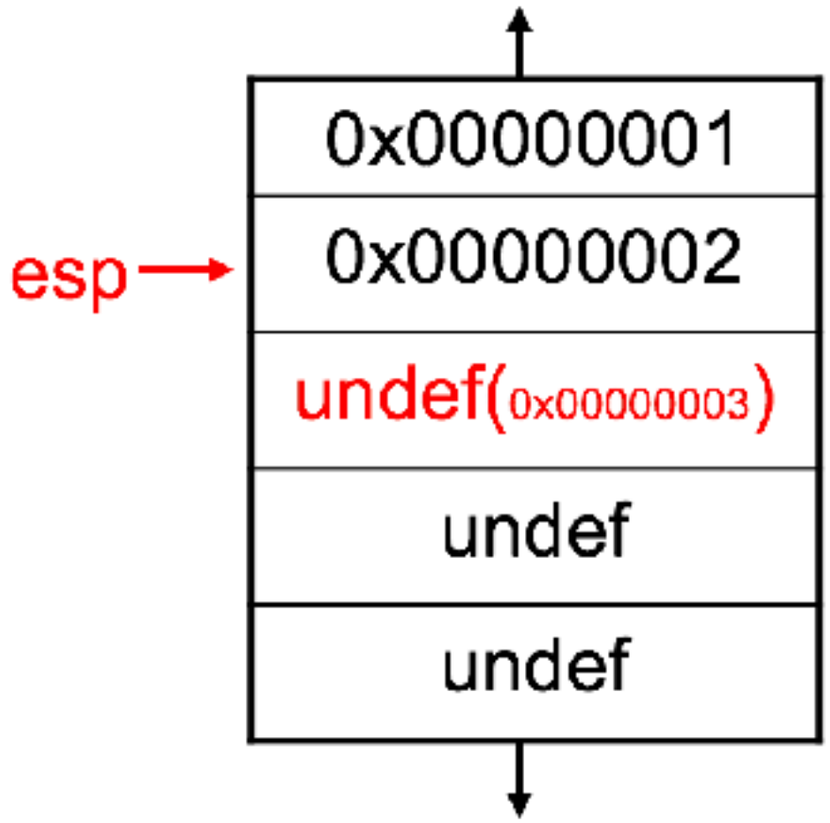
Registers After

eax	0x00000003
esp	0x0012FF8C

Stack Before



Stack After



CALL - CALL PROCEDURE

- CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
  - For use by RET for when the procedure is done
- Then it changes eip to the address given in the instruction
- Destination address can be specified in multiple ways
  - Absolute address
  - Relative address (relative to the end of the instruction)

RET – RETURN FROM PROCEDURE

- Two forms
  - –Pop the top of the stack into eip (remember pop increments stack pointer)
    - In this form, the instruction is just written as “ret”
  - –Pop the top of the stack into eip and add a constant number of bytes to esp
    - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc

MOV – MOVE INSTRUCTION

- Can move:
  - register to register
  - memory to register, register to memory
  - immediate to register, immediate to memory
- Never memory to memory!
- Memory addresses are given in r/m32 form talked about later

EXAMPLE – 1

The stack frames in this example will be very simple.
Only saved frame pointer (ebp) and saved return addresses (eip).

```
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}
int main(){
    sub();
    return 0xf00d;
}
```

```
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
```

EIP = 00401010, but no instruction yet executed

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF6C ⌘

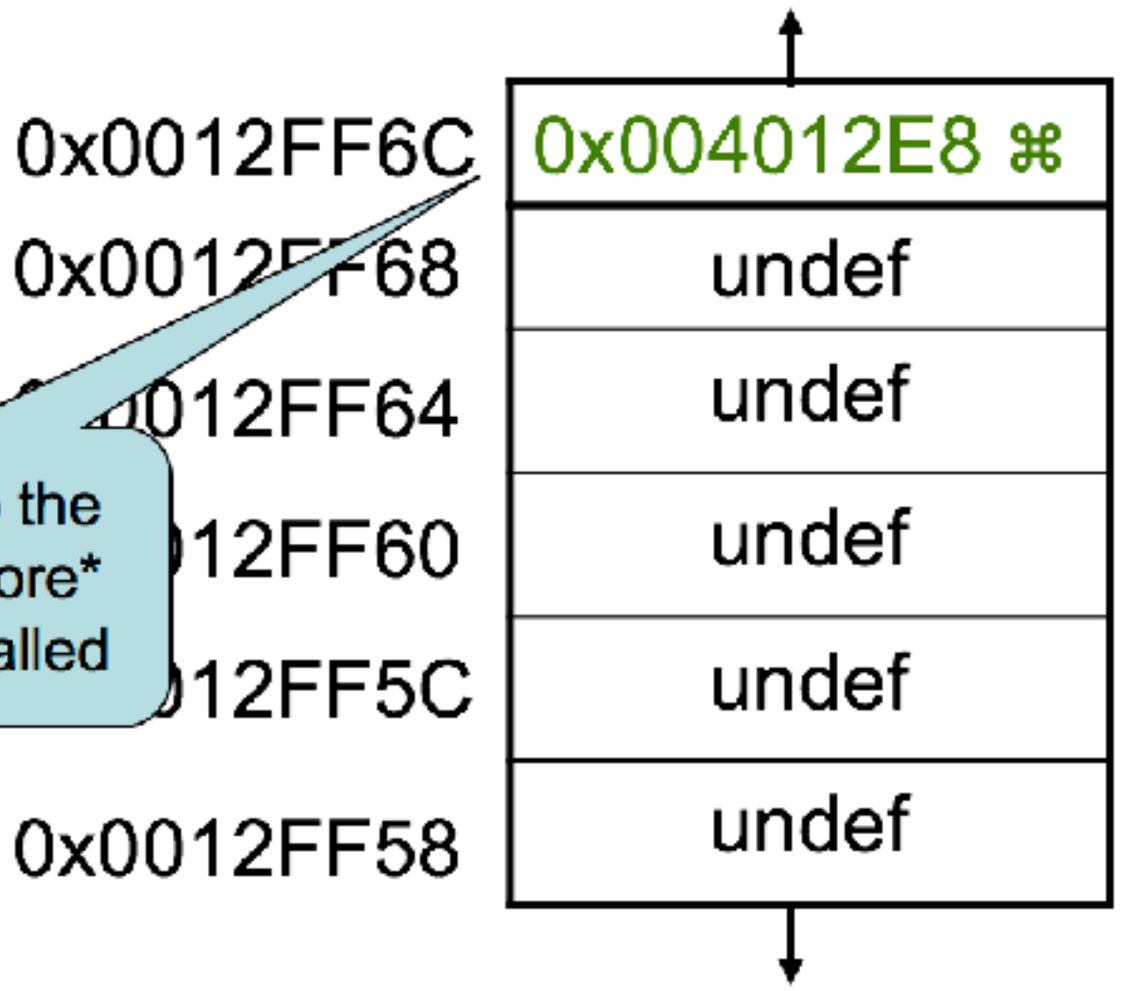
Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
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
```



eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF68 ♍

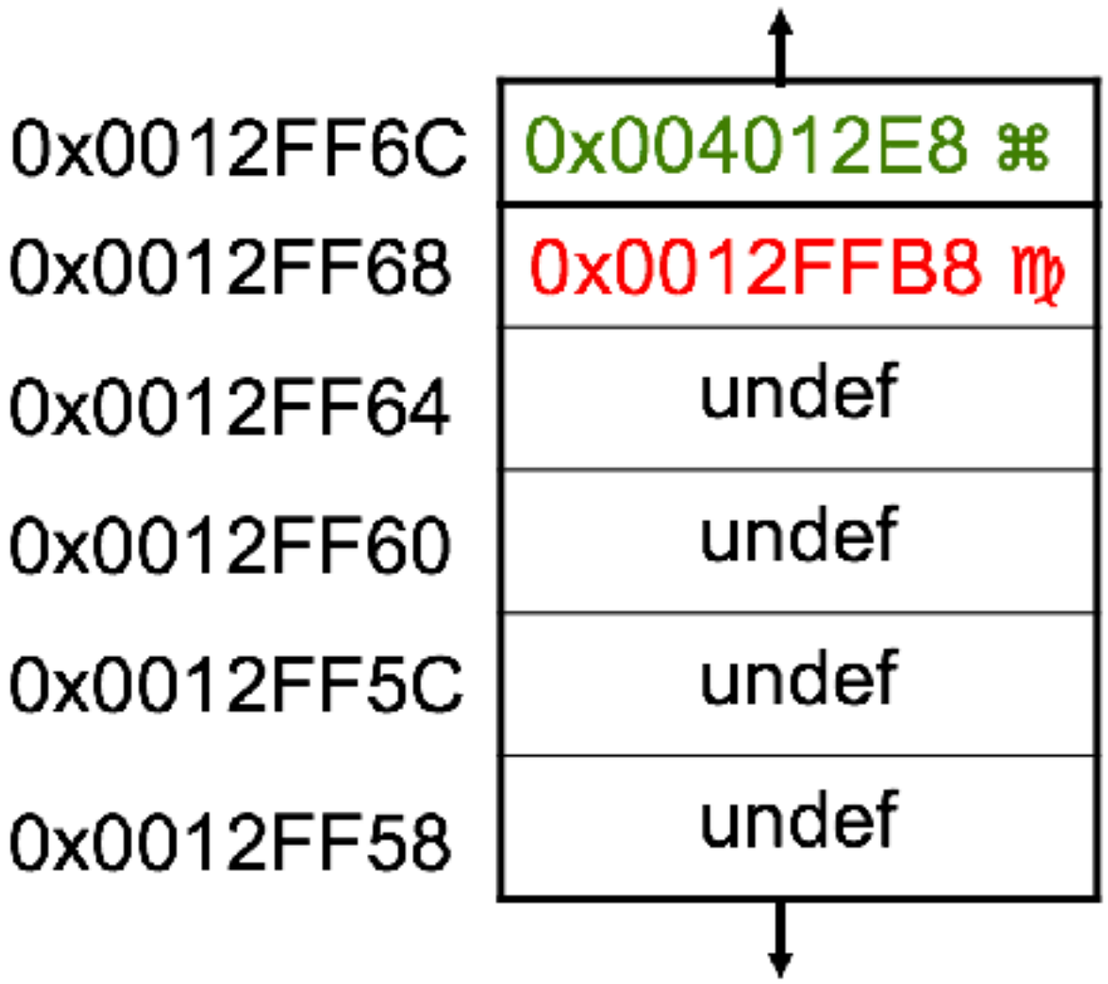
Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

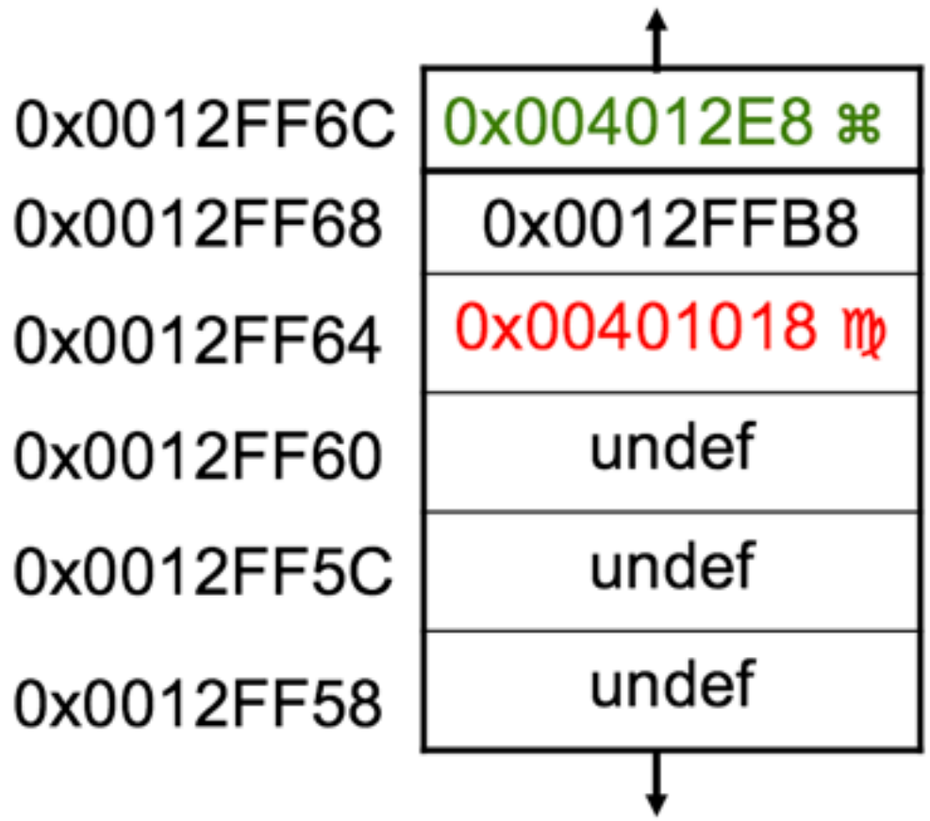
```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp   ☒
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```



eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF64 ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

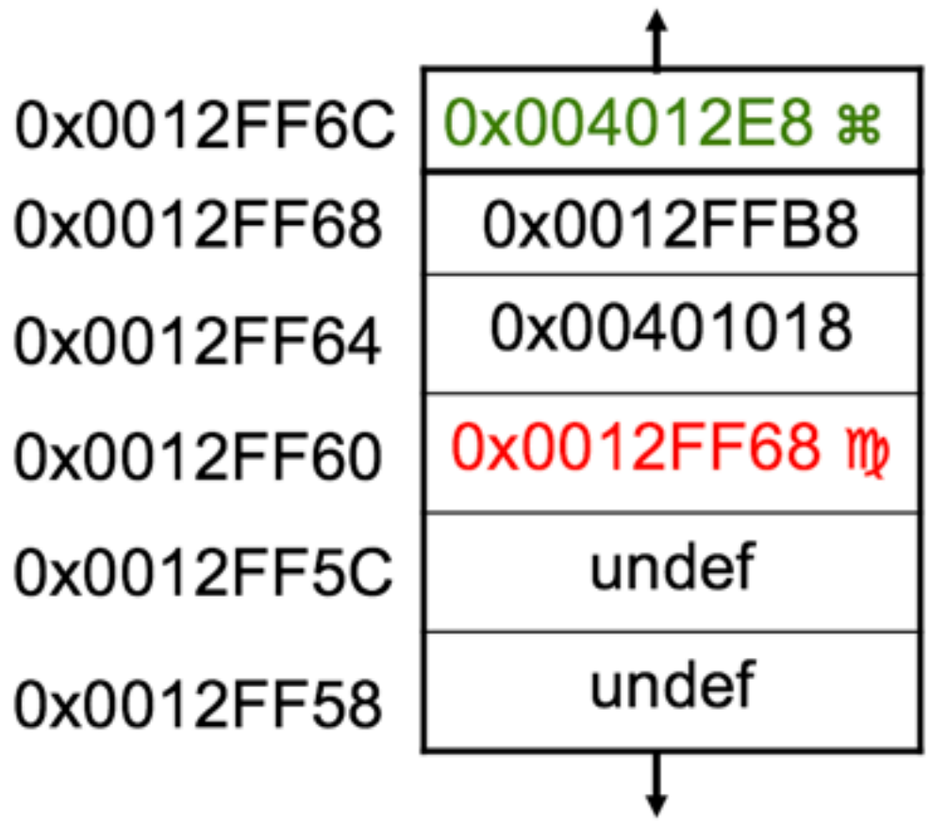
```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h) ☒
00401018  mov         eax,0F00Dh
0040101D  pop         ebp
0040101E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



eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF60 ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

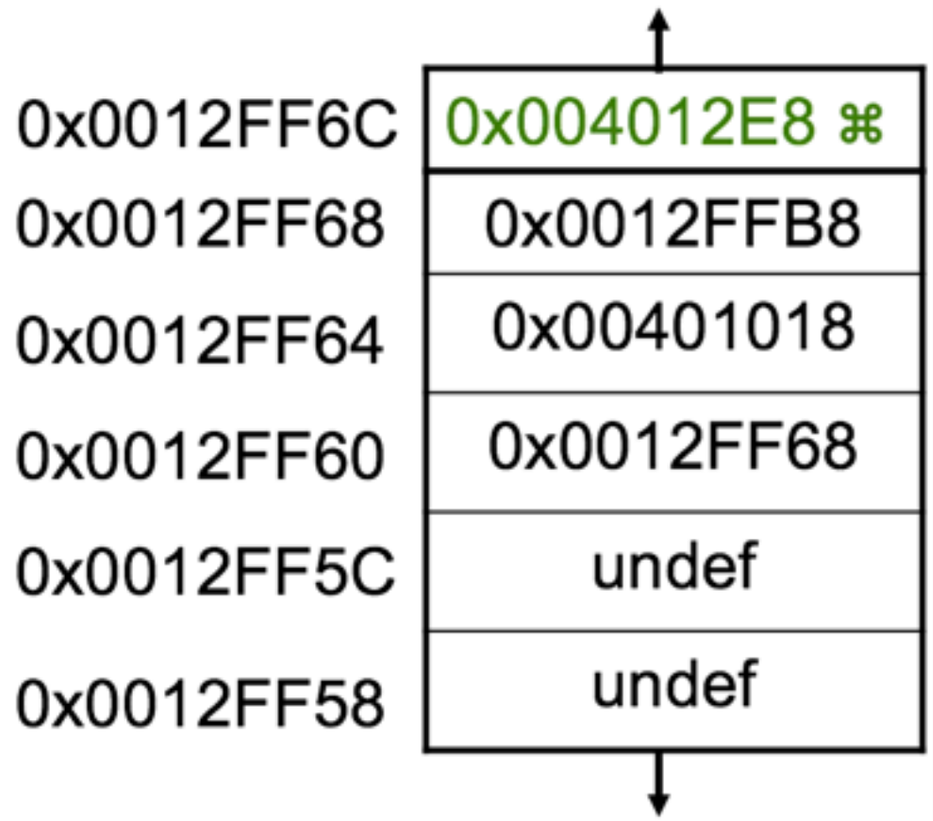
```
sub:
00401000  push      ebp ☒
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```



eax	0x003435C0 ⌘
ebp	0x0012FF60 ♍
esp	0x0012FF60

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

```
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
```

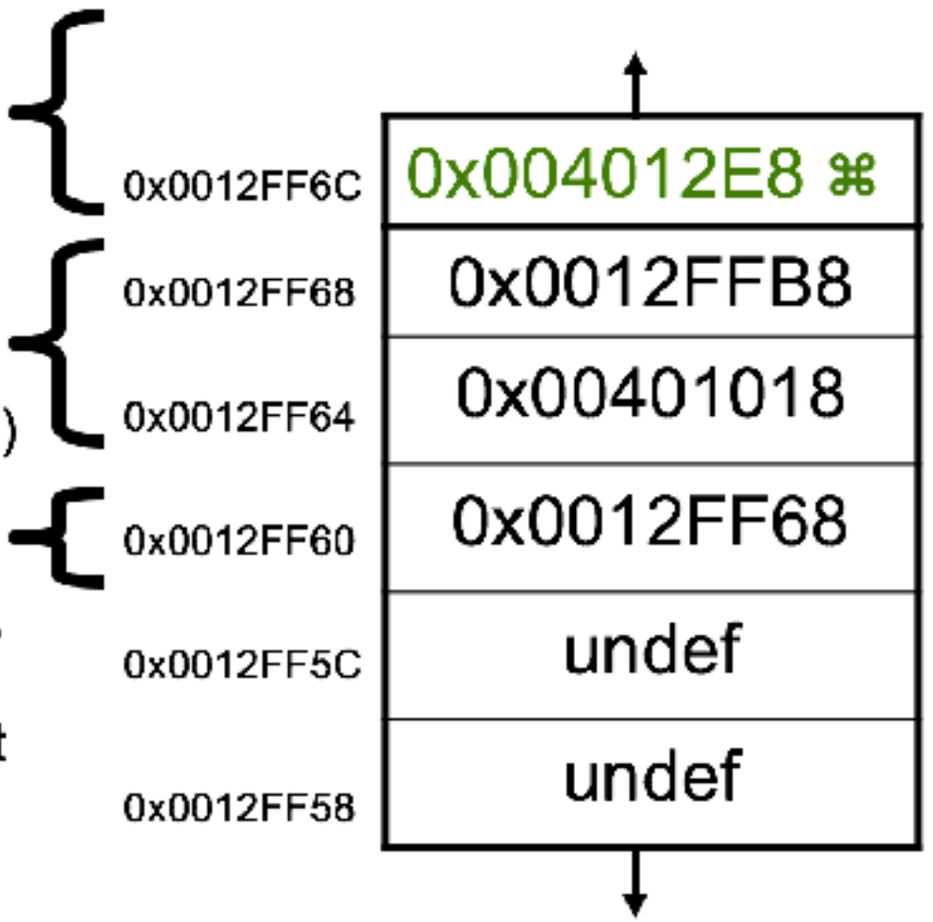


```
sub
push    ebp
mov     ebp, esp
mov     eax, 0BEEFh
pop     ebp
retn
main
push    ebp
mov     ebp, esp
call    _sub
mov     eax, 0F00Dh
pop     ebp
retn
```

"Function-before-main"'s frame

main's frame
(saved frame pointer and saved return address)

sub's frame
(only saved frame pointer, because it doesn't call anything else, and doesn't have local variables)



eax	0x0000BEEF
ebp	0x0012FF60
esp	0x0012FF60

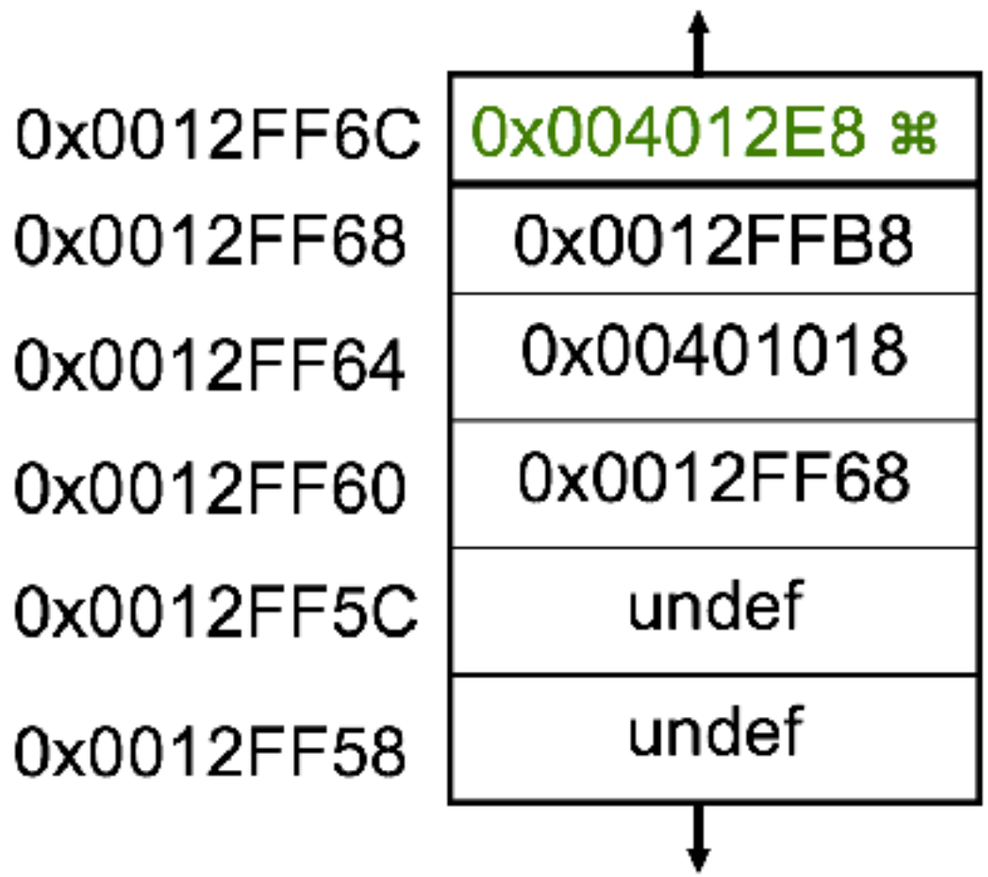
```
sub:
00401000   push        ebp
00401001   mov         ebp,esp
00401003   mov         eax,0BEEFh ☒
00401008   pop         ebp
00401009   ret
main:
00401010   push        ebp
00401011   mov         ebp,esp
00401013   call        sub (401000h)
00401018   mov         eax,0F00Dh
0040101D   pop      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

Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**



eax	0x0000BEEF
ebp	0x0012FF68 ♍
esp	0x0012FF64 ♍

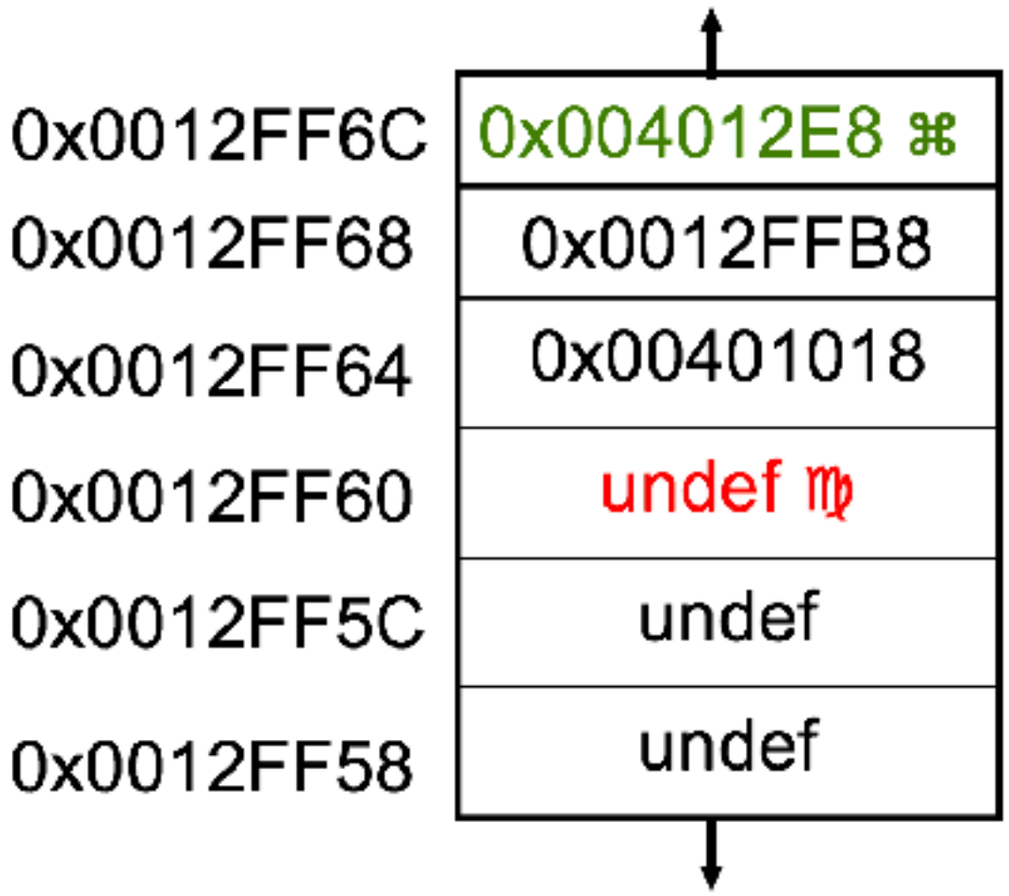
```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp ☒
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

Key:

☒ **executed instruction,**

♍ **modified value**

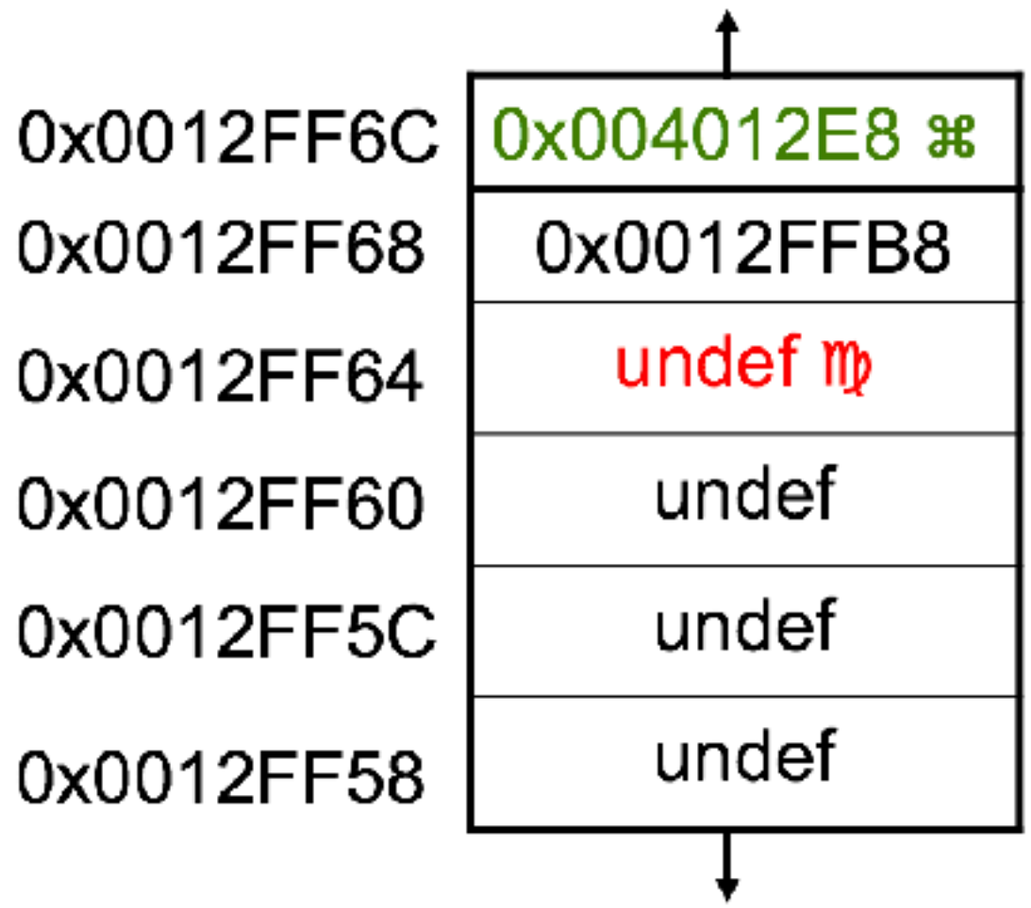
⌘ **start value**



eax	0x0000BEEF
ebp	0x0012FF68
esp	0x0012FF68 ♍

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

```
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
```



eax	0x0000F00D ♍
ebp	0x0012FF68
esp	0x0012FF68

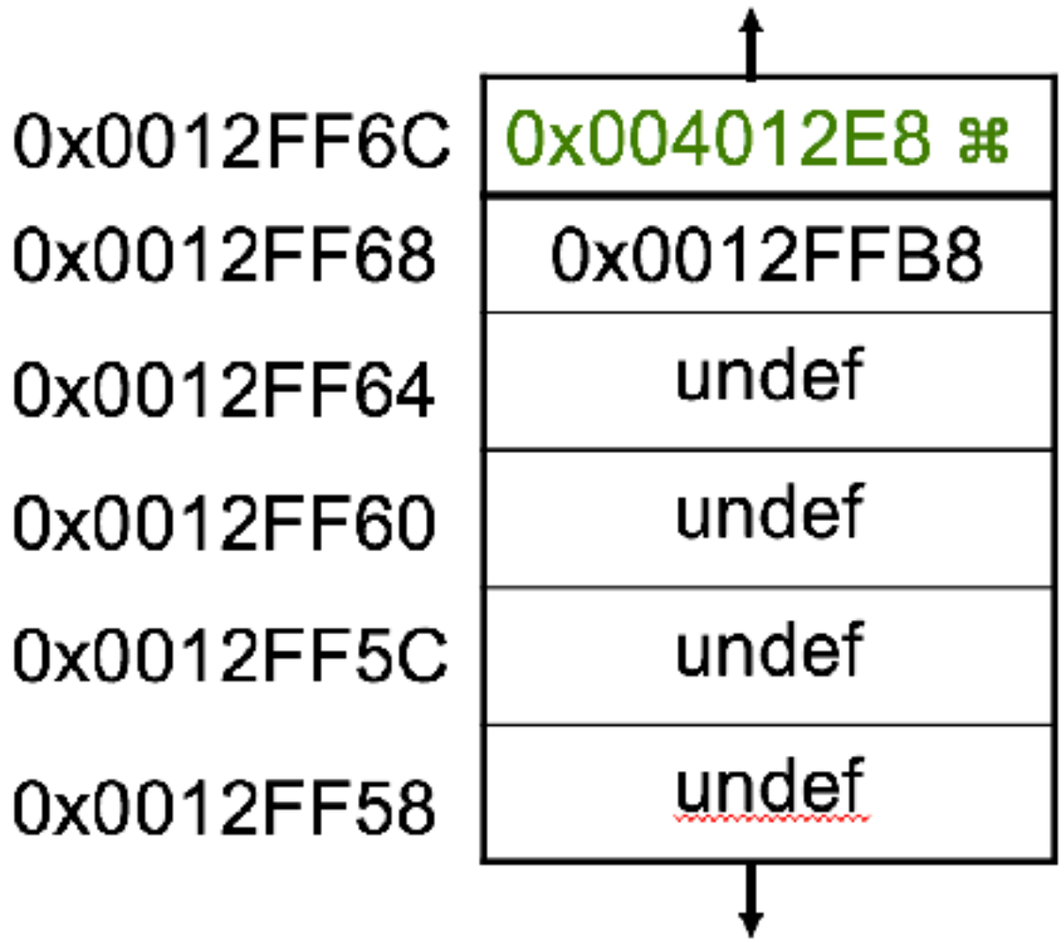
```
sub:
00401000   push        ebp
00401001   mov         ebp,esp
00401003   mov         eax,0BEEFh
00401008   pop         ebp
00401009   ret
main:
00401010   push        ebp
00401011   mov         ebp,esp
00401013   call        sub (401000h)
00401018   mov         eax,0F00Dh ☒
0040101D   pop      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

Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**



eax	0x0000F00D
ebp	0x0012FFB8 ℳ
esp	0x0012FF6C ℳ

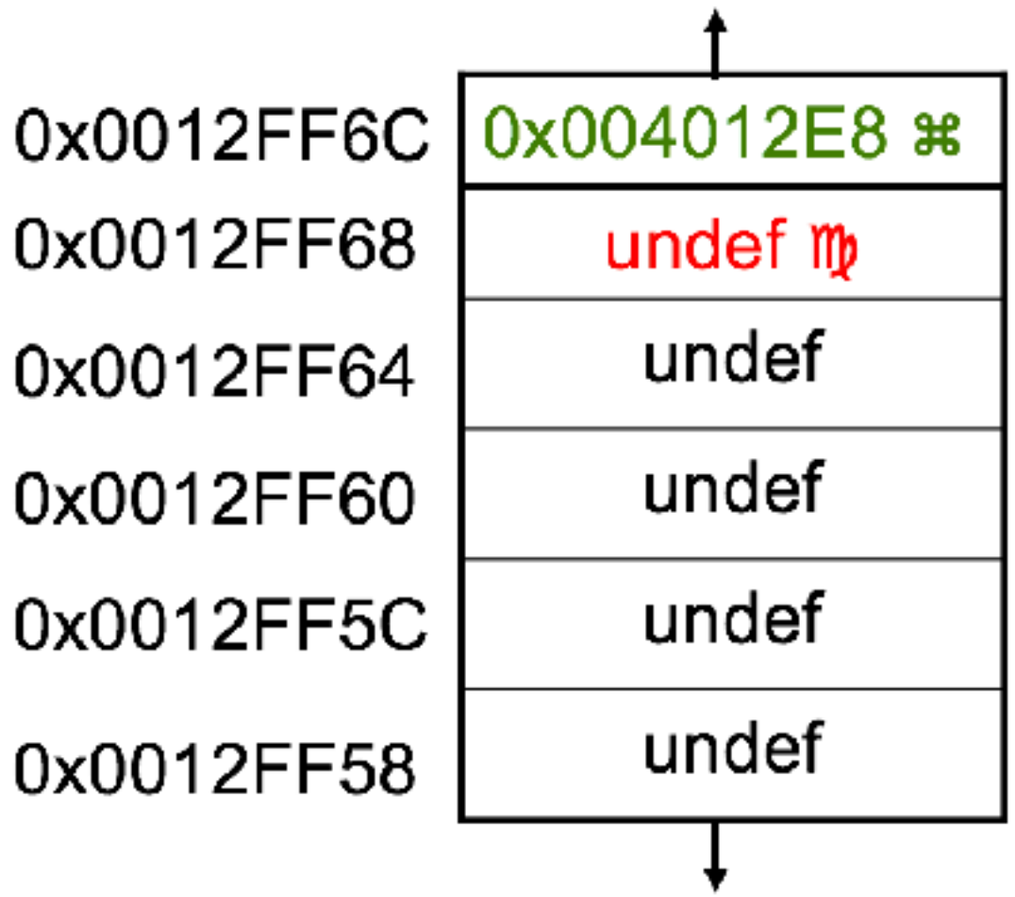
Key:

☒ **executed instruction,**

ℳ **modified value**

⌘ **start value**

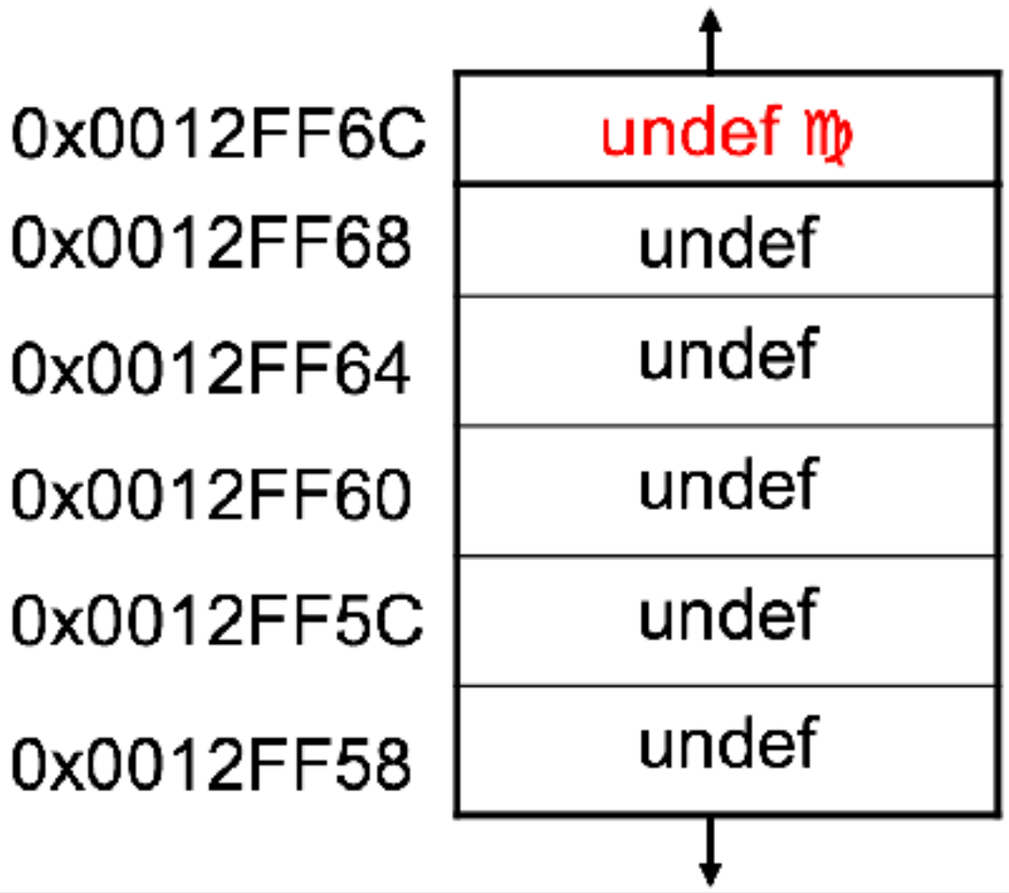
```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh
0040101D  pop         ebp ☒
0040101E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



eax	0x0000F00D
ebp	0x0012FFB8
esp	0x0012FF70 ♍

Key:
☒ **executed instruction,**
♍ **modified value**
⌘ **start value**

```
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 ☒
```



THAT'S ALL FOR TODAY

QUESTIONS ???