Sara Palacios Chavarro
Lab 6
Análisis Forense
Universidad del Rosario

Read the introduction of the section 6 "Recognizing C Code Constructs in Assembly" and explain what means a "Code Construct". What aspects may impact the way as assembly code is generated?

- A code construct is a code abstraction level that defines a functional property but not the details of its implementation. The compiler is the main factor that can create differences between assembly codes, because compiler versions and settings can impact how a particular construct apperars in disassembly.

Read the section "Gobal vs Local Variables" and identify what are the differences in the compilation of a code that employs global vs one that employs local Variables.

- Gobal variables can be accessed and used by any function in a program, they are referenced by memry addresses. Local variables can be accessed only by the function in which they are defined and are referenced by the stack addresses.

- Global variable IDA result: The executed code was:

```c
#include<stdio.h>

int x=1;
int y=2;

void main()
{
  x = x+y;
  printf("Total = %d\n", x);
}
```

In the information generated by IDA we can see that the global variable x is signified by dword_404004, which is a memory location at 0x404004. Notice that this variable is changed in memory when eax is moved into dword_404004.

```
.text:00401410
.text:00401410 sub_401410      proc near               ; CODE XREF: sub_4011A0+8E↑p
.text:00401410
.text:00401410 var_10          = dword ptr -10h
.text:00401410 var_C           = dword ptr -0Ch
.text:00401410
.text:00401410                 push    ebp
.text:00401411                 mov     ebp, esp
.text:00401413                 and     esp, 0FFFFFFF0h
.text:00401416                 sub     esp, 10h        ; char *
.text:00401419                 call    sub_4019A0
.text:0040141E                 mov     edx, dword_404004
.text:00401424                 mov     eax, dword_404008
.text:00401429                 add     eax, edx
.text:0040142B                 mov     dword_404004, eax
.text:00401430                 mov     eax, dword_404004
.text:00401435                 mov     [esp+10h+var_C], eax
.text:00401439                 mov     [esp+10h+var_10], offset aTotalD ; "Total = %d\n"
.text:00401440                 call    printf
.text:00401445                 nop
.text:00401446                 leave
.text:00401447                 retn
.text:00401447 sub_401410      endp
.text:00401447
.text:00401447 ; ---------------------------------------------------------------------------
```

- Local variable IDA result: The code used was:

```c
#include<stdio.h>

void main()
{
  int x=1;
  int y=2;

  x = x+y;
  printf("Total = %d\n", x);
}
```

The local variable x is located on the stack at a constant offset relative to ebp. Then it is changed to esp in order to define the memory location [esp+20h+var$_4$] which is used consistenly thoughout this function to reference the local variable x. This tell us that this memory location is a stack-based local variable that is references only in this function .

```
.text:00401410
.text:00401410 sub_401410      proc near               ; CODE XREF: sub_4011A0+8E↑p
.text:00401410
.text:00401410 var_20          = dword ptr -20h
.text:00401410 var_1C          = dword ptr -1Ch
.text:00401410 var_8           = dword ptr -8
.text:00401410 var_4           = dword ptr -4
.text:00401410
.text:00401410                 push    ebp
.text:00401411                 mov     ebp, esp
.text:00401413                 and     esp, 0FFFFFFF0h
.text:00401416                 sub     esp, 20h        ; char *
.text:00401419                 call    sub_4019A0
.text:0040141E                 mov     [esp+20h+var_4], 1
.text:00401426                 mov     [esp+20h+var_8], 2
.text:0040142E                 mov     eax, [esp+20h+var_8]
.text:00401432                 add     [esp+20h+var_4], eax
.text:00401436                 mov     eax, [esp+20h+var_4]
.text:0040143A                 mov     [esp+20h+var_1C], eax
.text:0040143E                 mov     [esp+20h+var_20], offset aTotalD ; "Total = %d\n"
.text:00401445                 call    printf
.text:0040144A                 nop
.text:0040144B                 leave
.text:0040144C                 retn
.text:0040144C sub_401410      endp
```

Read the section "Disassembling Arithmetic Operations" and explain to your classmates how the operations (addition, substraction, increment, decrement and modulo) are represented in assembly code.

- Arithmetic Operations: The executed code was:

```c
#include<stdio.h>

void main(){
    int a = 0;
    int b = 1;
    a = a + 11;
    a = a - b;
    a--;
    b++;
    b = a % 3;
    printf("resultado = %d\n", b);

}
```

```
.text:00401410                 push    ebp
.text:00401411                 mov     ebp, esp
.text:00401413                 and     esp, 0FFFFFFF0h
.text:00401416                 sub     esp, 20h        ; char *
.text:00401419                 call    sub_4019D0
.text:0040141E                 mov     [esp+20h+var_4], 0
.text:00401426                 mov     [esp+20h+var_8], 1
.text:0040142E                 add     [esp+20h+var_4], 0Bh
.text:00401433                 mov     eax, [esp+20h+var_8]
.text:00401437                 sub     [esp+20h+var_4], eax
.text:0040143B                 sub     [esp+20h+var_4], 1
.text:00401440                 add     [esp+20h+var_8], 1
.text:00401445                 mov     ecx, [esp+20h+var_4]
.text:00401449                 mov     edx, 55555556h
.text:0040144E                 mov     eax, ecx
.text:00401450                 imul    edx
.text:00401452                 mov     eax, ecx
.text:00401454                 sar     eax, 1Fh
.text:00401457                 sub     edx, eax
.text:00401459                 mov     eax, edx
.text:0040145B                 add     eax, eax
.text:0040145D                 add     eax, edx
.text:0040145F                 sub     ecx, eax
.text:00401461                 mov     eax, ecx
.text:00401463                 mov     [esp+20h+var_8], eax
```

We can see that there are local variables since there is a push to the stack. The variable a is called var_ 4 located at [esp+20h+var_ 4] and is initialized to 0, the variable b is called var_ 8 located at [esp+20h+var_ 8] and is initialized to 1. Then the different instructions are executed.

Read the section "Recognizing if Statements" and explain to your classmates how to recognize an if/else structure in assembly code.

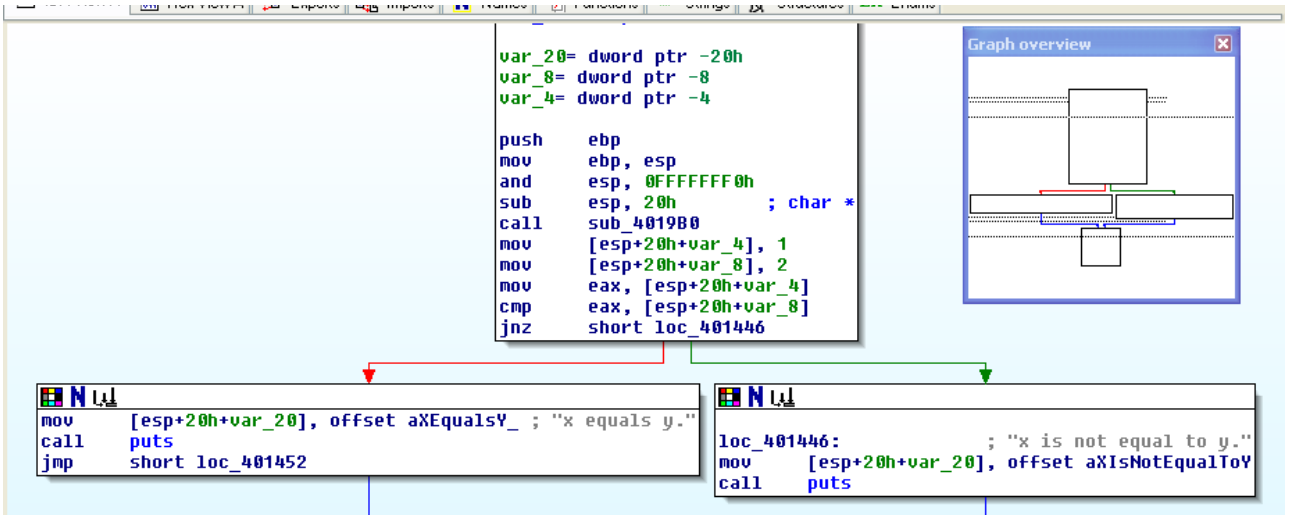
- The executed code was:

```c
#include<stdio.h>

void main(){

    int x = 1;
    int y = 2;
    if(x == y){
    printf("x equals y.\n");
    }else{
    printf("x is not equal to y.\n");
    }
}
```

- Since it is an if statement there are two options to evaluate, the diagram show us this options. First the program creates the local variables and then it checks the conditions and according to its decision the result is given at the end. The main functions here are the jmp or jnz which let the program decide which route it need to go through.



Read the section "Recognizing Nested if Statements" and explain to your classmates how to recognize a "Nested IF" structure in assembly code.

- The executed code was:

```c
#include<stdio.h>

void main(){
    int x = 0;
    int y = 1;
    int z = 2;
    if(x == y){
        if(z==0){
            printf("z is zero and x = y.\n");
        }else{
            printf("z is non-zero and x = y.\n");
        }
    }else{
        if(z==0){
            printf("z zero and x != y.\n");
        }else{
            printf("z non-zero and x != y.\n");
        }
    }
}
```

- As the if statement, the nested if evaluates blocks, so in order to recognize this statements you need to look for the blocks that are going to be evaluated. Of course the local variables are initialized and the first steps are done before the this part of the execution, but the most important parts are these blocks, as in the normal if statement the jmp and jnz need to be present.

```
.text:00401455 ; ---------------------------------------------------------------------------
.text:00401455
.text:00401455 loc_401455:                             ; CODE XREF: sub_401410+35↑j
.text:00401455                 mov     [esp+20h+var_20], offset aZIsNonZeroAndX ; "z is non-zero and x
.text:0040145C                 call    puts
.text:00401461                 jmp     short loc_401484
.text:00401463 ; ---------------------------------------------------------------------------
.text:00401463
.text:00401463 loc_401463:                             ; CODE XREF: sub_401410+2E↑j
.text:00401463                 cmp     [esp+20h+var_C], 0
.text:00401468                 jnz     short loc_401478
.text:0040146A                 mov     [esp+20h+var_20], offset aZZeroAndXY_ ; "z zero and x != y."
.text:00401471                 call    puts
.text:00401476                 jmp     short loc_401484
.text:00401478 ; ---------------------------------------------------------------------------
.text:00401478
.text:00401478 loc_401478:                             ; CODE XREF: sub_401410+58↑j
.text:00401478                 mov     [esp+20h+var_20], offset aZNonZeroAndXY_ ; "z non-zero and x !=
.text:0040147F                 call    puts
.text:00401484
.text:00401484 loc_401484:                             ; CODE XREF: sub_401410+43↑j
.text:00401484                                         ; sub_401410+51↑j ...
.text:00401484                 nop
.text:00401485                 leave
.text:00401486                 retn
```

Read the section "Recognizing Loops" and explain to your classmates how to recognize a FOR structure in assembly code.

- The executed code was:

```c
#include<stdio.h>

void main(){
    int i;
    for(i=0; i<100; i++){
        printf("i equals %d\n", i);
    }
}
```

- We can see that after the initialization, the program is jumping to loc_401441 which is the start of the for loop. It evaluates the condition or the counter for the loop, if it fits then it executes the respective instructions that are at loc_401428.

```
.text:00401410
.text:00401410 var_20          = dword ptr -20h
.text:00401410 var_1C          = dword ptr -1Ch
.text:00401410 var_4           = dword ptr -4
.text:00401410
.text:00401410                 push    ebp
.text:00401411                 mov     ebp, esp
.text:00401413                 and     esp, 0FFFFFFF0h
.text:00401416                 sub     esp, 20h        ; char *
.text:00401419                 call    sub_4019A0
.text:0040141E                 mov     [esp+20h+var_4], 0
.text:00401426                 jmp     short loc_401441
.text:00401428 ; ---------------------------------------------------------------------------
.text:00401428
.text:00401428 loc_401428:                             ; CODE XREF: sub_401410+36↓j
.text:00401428                 mov     eax, [esp+20h+var_4]
.text:0040142C                 mov     [esp+20h+var_1C], eax
.text:00401430                 mov     [esp+20h+var_20], offset aIEqualsD ; "i equals %d\n"
.text:00401437                 call    printf
.text:0040143C                 add     [esp+20h+var_4], 1
.text:00401441
.text:00401441 loc_401441:                             ; CODE XREF: sub_401410+16↑j
.text:00401441                 cmp     [esp+20h+var_4], 63h
.text:00401446                 jle     short loc_401428
.text:00401448                 nop
```

Read the section "Recognizing Loops" and explain to your classmates how to recognize a WHILE structure in assembly code.

- The executed code was:

```c
#include<stdio.h>

void main(){
    int status=0;
    int result = 0;
    while(status == 0){
        result = performAction();
        status = checkResult(result);
    }

}
```

- Just like the previous loop. The initialization is done befor the loop evaluation, since this is a while we need a condition to evaluate, therefor there must be a jmp instruction. This is jumping to loc_40145D if it is true then ir jumps to the set of instructions inside of the loop.

```
.text:0040142D                 call    sub_4019D0
.text:00401432                 mov     [esp+20h+var_4], 0
.text:0040143A                 mov     [esp+20h+var_8], 0
.text:00401442                 jmp     short loc_40145D
.text:00401444 ; ---------------------------------------------------------------------------
.text:00401444
.text:00401444 loc_401444:                             ; CODE XREF: sub_401424+3E↓j
.text:00401444                 call    sub_401410
.text:00401449                 mov     [esp+20h+var_8], eax
.text:0040144D                 mov     eax, [esp+20h+var_8]
.text:00401451                 mov     [esp+20h+var_20], eax
.text:00401454                 call    sub_40141A
.text:00401459                 mov     [esp+20h+var_4], eax
.text:0040145D
.text:0040145D loc_40145D:                             ; CODE XREF: sub_401424+1E↑j
.text:0040145D                 cmp     [esp+20h+var_4], 0
.text:00401462                 jz      short loc_401444
.text:00401464                 mov     [esp+20h+var_20], offset aStatus ; "status "
.text:0040146B                 call    printf
.text:00401470                 nop
.text:00401471                 leave
.text:00401472                 retn
.text:00401472 sub_401424      endp
.text:00401472
.text:00401472 ; ---------------------------------------------------------------------------
```