CSec15233
Malicious Software Analysis

Recognizing C Code Constructs in Assembly

Qasem Abu Al-Haija

Motivation

- Successful reverse engineers do not evaluate each instruction individually unless they must.
  - The process is just too tedious, and
  - Too many instructions!

- As a malware analyst, you must be able to obtain a high-level picture of code functionality by analyzing instructions as groups
  - focusing on individual instructions only as needed.

- This skill takes time to develop.

Malware Code

- Malware is typically developed using an HLL, most commonly C.
- A code construct is a code abstraction level that defines a functional property but not the details of its implementation.
  - Loops, if statements, linked lists, switch statements,... etc.
- Programs can be broken down into individual constructs
  - when combined, implement the overall functionality of the program
- Your goal will be to go from disassembly to H.L. constructs
  - Compiler versions and settings can impact how a particular construct appears in disassembly

Global Variables

- accessed and used by any function in a program.
  - Example: Following listing defines x and y variables outside the function.

```
int x = 1;
int y = 2;

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

Listing 6-1: A simple program with two global variables

```
00401003    mov     eax, dword_40CF60
00401008    add     eax, dword_40C000
0040100E    mov     dword_40CF60, eax ❶
00401013    mov     ecx, dword_40CF60
00401019    push    ecx
0040101A    push    offset aTotalD  ;"total = %d\n"
0040101F    call    printf
```

Listing 6-3: Assembly code for the global variable example in Listing 6-1

- In Listing 6-3, the global variable x is signified by dword_40CF60, a memory location at **0x40CF60**.
- Notice that x is changed in memory when eax is moved into dword_40CF60 at ① .
- All subsequent functions that utilize this variable will be impacted.

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main          proc near           ; CODE XREF: ___tmainCRTStartup+10Ap
.text:00401000
.text:00401000 argc           = dword ptr  8
.text:00401000 argv           = dword ptr  0Ch
.text:00401000 envp           = dword ptr  10h
.text:00401000
.text:00401000                push    ebp
.text:00401001                mov     ebp, esp
.text:00401003                mov     eax, dword_403018
.text:00401008                add     eax, dword_40301C
.text:0040100E                mov     dword_403018, eax
.text:00401013                mov     ecx, dword_403018
.text:00401019                push    ecx
.text:0040101A                push    offset aTotalD  ; "Total = %d\n"
.text:0040101F                call    printf
.text:00401024                add     esp, 8
.text:00401027                xor     eax, eax
.text:00401029                pop     ebp
.text:0040102A                retn
.text:0040102A _main          endp
```

Local Variables

- accessed only by the function in which they are Defined
  - Example: Following listing defines x and y variables within the function.

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

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

Listing 6-2: A simple program with two local variables

```
00401006        mov     dword ptr [ebp-4], 1
0040100D        mov     dword ptr [ebp-8], 2
00401014        mov     eax, [ebp-4]
00401017        add     eax, [ebp-8]
0040101A        mov     [ebp-4], eax
0040101D        mov     ecx, [ebp-4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

Listing 6-4: Assembly code for the local variable example in Listing 6-2, without labeling

- In Listings 6-4, the local variable x is located on the stack at a constant offset relative to ebp.
- Memory location [ebp-4] is used consistently throughout this function to reference the local variable x.
- This tells us that ebp-4 is a stack-based local variable that is referenced only in the function in which it is defined.

```
00401006    mov     [ebp+var_4], 1
0040100D    mov     [ebp+var_8], 2
00401014    mov     eax, [ebp+var_4]
00401017    add     eax, [ebp+var_8]
0040101A    mov     [ebp+var_4], eax
0040101D    mov     ecx, [ebp+var_4]
00401020    push    ecx
00401021    push    offset aTotalD  ; "total = %d\n"
00401026    call    printf
```

Listing 6-5: Assembly code for the local variable example shown in Listing 6-2, with labeling

- In Listing 6-5, x has been nicely labeled by IDA Pro Disassembler with the dummy name var_4.
    - Dummy names can be renamed to meaningful names that reflect their function.
- Having this local variable named var_4 instead of -4 simplifies your analysis.
    - because once you rename var_4 to x, you won't need to track the offset -4 in your head throughout the function.

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main        proc near            ; CODE XREF: ___tmainCRTStartup+10Ap
.text:00401000
.text:00401000 var_4        = dword ptr -4 ; int x
.text:00401000 var_8        = dword ptr -8 ; int y
.text:00401000 argc         = dword ptr  8
.text:00401000 argv         = dword ptr  0Ch
.text:00401000 envp         = dword ptr  10h
.text:00401000
.text:00401000              push   ebp
.text:00401001              mov    ebp, esp
.text:00401003              sub    esp, 8
.text:00401006              mov    [ebp+var_4], 1
.text:0040100D              mov    [ebp+var_8], 2
.text:00401014              mov    eax, [ebp+var_4]
.text:00401017              add    eax, [ebp+var_8]
.text:0040101A              mov    [ebp+var_8], eax
.text:0040101D              mov    ecx, [ebp+var_4]
.text:00401020              push   ecx
.text:00401021              push   offset aTotalD  ; "Total = %d\n"
.text:00401026              call   printf
.text:0040102B              add    esp, 8
.text:0040102E              xor    eax, eax
.text:00401030              mov    esp, ebp
.text:00401032              pop    ebp
.text:00401033              retn
.text:00401033 _main        endp
```

Arithmetic Operations

```
main() {
    int a = 0;
    int b = 1;

    a = a + 11;
    a = a - b;
    a--;
    b++;
    b = a % 3;
}
```

Listing 6-6: C code with two variables and a variety of arithmetic

```
00401006    mov     [ebp+var_4], 0
0040100D    mov     [ebp+var_8], 1
00401014    mov     eax, [ebp+var_4] ❶
00401017    add     eax, 0Bh
0040101A    mov     [ebp+var_4], eax
0040101D    mov     ecx, [ebp+var_4]
00401020    sub     ecx, [ebp+var_8] ❷
00401023    mov     [ebp+var_4], ecx
00401026    mov     edx, [ebp+var_4]
00401029    sub     edx, 1 ❸
0040102C    mov     [ebp+var_4], edx
0040102F    mov     eax, [ebp+var_8]
00401032    add     eax, 1 ❹
00401035    mov     [ebp+var_8], eax
00401038    mov     eax, [ebp+var_4]
0040103B    cdq
0040103C    mov     ecx, 3
00401041    idiv    ecx
00401043    mov     [ebp+var_8], edx ❺
```

Listing 6-7: Assembly code for the arithmetic example in Listing 6-6

- In this example, a and b are local variables because they are referenced by the stack.

- IDA Pro has labeled a as var_4 and b as var_8. First, var_4 and var_8 are initialized to 0 and 1, respectively. a is moved into eax ①, and then 0x0b is added to eax, thereby incrementing a by 11.

- b is then subtracted from a ②. (The compiler decided to use the sub and add instructions ③ and ④ instead of the inc and dec functions.)

- The final five assembly instructions implement the modulo. When performing the div or idiv instruction ⑤, you are dividing edx:eax by the operand and storing the result in eax and the remainder in edx. That is why edx is moved into var_8 ⑤.

- .text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
- .text:00401000 _main proc near ; CODE XREF: ___tmainCRTStartup+10Ap
- .text:00401000
- .text:00401000 var_8 = dword ptr -8
- .text:00401000 var_4 = dword ptr -4
- .text:00401000 argc = dword ptr 8
- .text:00401000 argv = dword ptr 0Ch
- .text:00401000 envp = dword ptr 10h
- .text:00401000
- .text:00401000 push ebp
- .text:00401001 mov ebp, esp
- .text:00401003 sub esp, 8
- .text:00401006 mov [ebp+var_4], 0
- .text:0040100D mov [ebp+var_8], 1
- .text:00401014 mov eax, [ebp+var_4]
- .text:00401017 add eax, 0Bh
- .text:0040101A mov [ebp+var_4], eax
- .text:0040101D mov ecx, [ebp+var_4]
- .text:00401020 sub ecx, [ebp+var_8]

- .text:00401023 mov [ebp+var_4], ecx
- .text:00401026 mov edx, [ebp+var_4]
- .text:00401029 sub edx, 1
- .text:0040102C mov [ebp+var_4], edx
- .text:0040102F mov eax, [ebp+var_8]
- .text:00401032 add eax, 1
- .text:00401035 mov [ebp+var_8], eax
- .text:00401038 mov eax, [ebp+var_4]
- .text:0040103B cdq
- .text:0040103C mov ecx, 3
- .text:00401041 idiv ecx ; EDX:EAX/ecx
- .text:00401043 mov [ebp+var_8], edx ; remainder
- .text:00401046 xor eax, eax
- .text:00401048 mov esp, ebp
- .text:0040104A pop ebp
- .text:0040104B retn
- .text:0040104B _main endp

Recognizing if Statements

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

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

Listing 6-8: C code `if` *statement example*

```
00401006        mov     [ebp+var_4], 1
0040100D        mov     [ebp+var_8], 2
00401014        mov     eax, [ebp+var_4]
00401017        cmp     eax, [ebp+var_8] ❶
0040101A        jnz     short loc_40102B ❷
0040101C        push    offset aXEqualsY_ ; "x equals y.\n"
00401021        call    printf
00401026        add     esp, 4
00401029        jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B        push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030        call    printf
```

Listing 6-9: Assembly code for the if statement example in Listing 6-8

Notice the conditional jump jnz at ②. There must be a conditional jump for an if statement, but not all conditional jumps correspond to if statements.

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main        proc near            ; CODE XREF: ___tmainCRTStartup+10Ap
.text:00401000
.text:00401000 var_4         = dword ptr -4 ; int x
.text:00401000 var_8         = dword ptr -8 ; int y
.text:00401000 argc          = dword ptr  8
.text:00401000 argv          = dword ptr  0Ch
.text:00401000 envp          = dword ptr  10h
.text:00401000
.text:00401000          push    ebp
.text:00401001          mov     ebp, esp
.text:00401003          sub     esp, 8
.text:00401006          mov     [ebp+var_4], 1
.text:0040100D          mov     [ebp+var_8], 2
.text:00401014          mov     eax, [ebp+var_4]
.text:00401017          cmp     eax, [ebp+var_8]
.text:0040101A          jnz     short loc_40102C
.text:0040101C          push    offset aXEqualsY_ ; "x equals y.\n"
.text:00401021          call    ds:printf
.text:00401027          add     esp, 4
.text:0040102A          jmp     short loc_40103A
.text:0040102C ; ---------------------------------------------------------------------------
.text:0040102C
.text:0040102C loc_40102C:                       ; CODE XREF: _main+1Aj
.text:0040102C          push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
.text:00401031          call    ds:printf
.text:00401037          add     esp, 4
.text:0040103A
.text:0040103A loc_40103A:                       ; CODE XREF: _main+2Aj
.text:0040103A          xor     eax, eax
.text:0040103C          mov     esp, ebp
.text:0040103E          pop     ebp
.text:0040103F          retn
.text:0040103F _main        endp
```

Analyzing Functions Graphically with IDA Pro

- IDA Pro has a graphing tool that is useful in recognizing constructs.
  - This feature is the default view for analyzing functions.

- Figure 6-1 shows a graph of assembly code example in Listing 6-9.
  - As you can see, two different paths (① and ②) of code execution led to the end of the function, and each path prints a different string.
  - Code path ① will print "x equals y.", and ② will print "x is not equal to y.“
  - IDA Pro adds false ① and true ② labels at the decision points at the bottom of the upper code box.
  - As you can imagine, graphing a function can greatly speed up the reverse-engineering process.

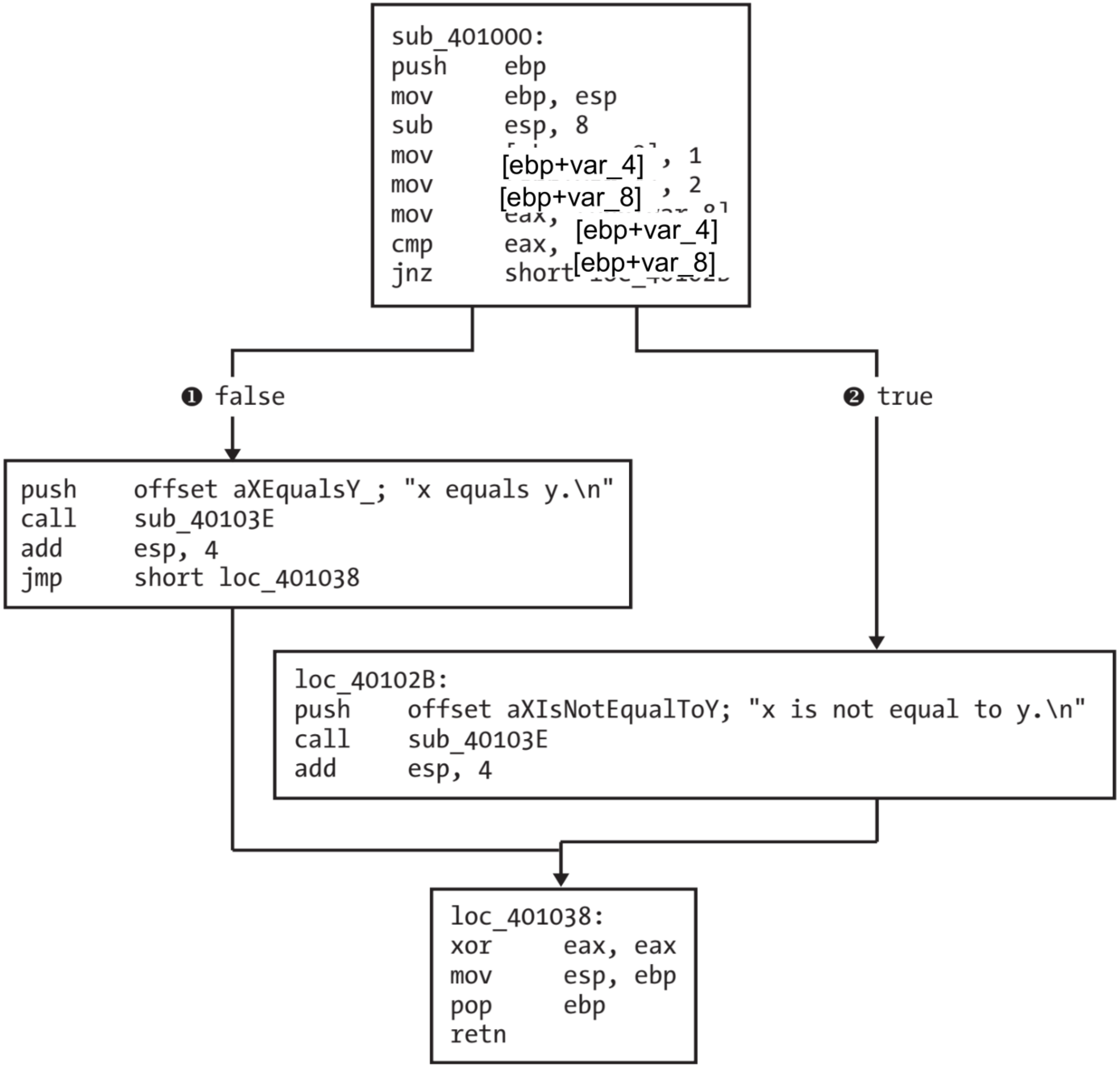
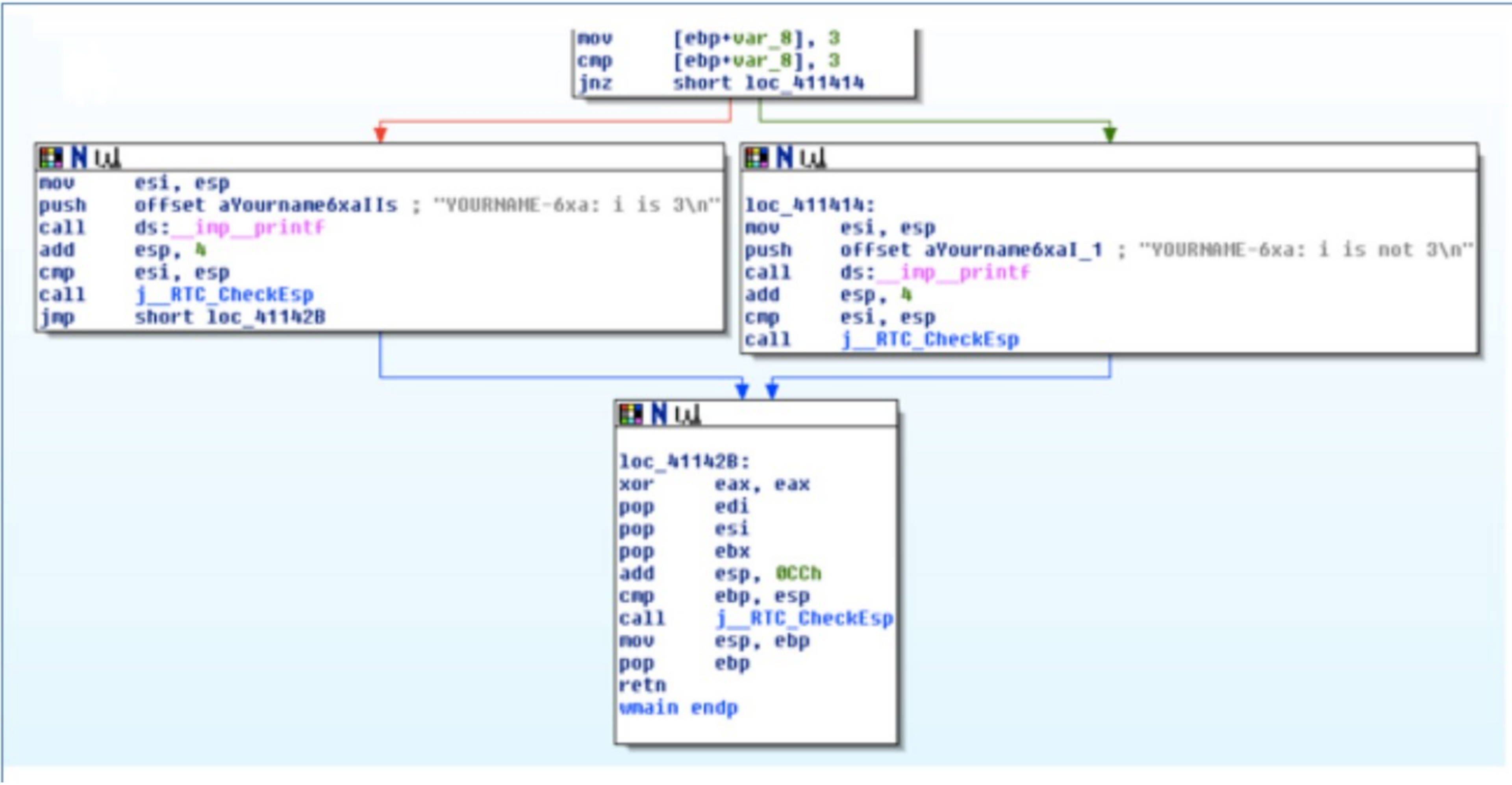


Figure 6-1: Disassembly graph for the `if` *statement example in Listing 6-9*


mov [ebp+var_8], 3
cmp [ebp+var_8], 3
jnz short loc_411414
mov esi, esp
push offset aYourname6xaIIs ; "YOURNAME-6xa: i is 3\n"
call ds:__imp__printf
add esp, 4
cmp esi, esp
call j__RTC_CheckEsp
jmp short loc_41142B
loc_411414:
mov esi, esp
push offset aYourname6xaI_1 ; "YOURNAME-6xa: i is not 3\n"
call ds:__imp__printf
add esp, 4
cmp esi, esp
call j__RTC_CheckEsp
loc_41142B:
xor eax, eax
pop edi
pop esi
pop ebx
add esp, 0CCh
cmp ebp, esp
call j__RTC_CheckEsp
mov esp, ebp
pop ebp
retn
wmain endp

Recognizing Nested if Statements

```
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");
        }
    }
}
```

- This is like Listing 6-8, except that two additional if statements are added within the original if statement, and both test to determine whether z is equal to 0.
- Despite this minor change to the C code, the assembly code is more complicated, as shown in Listing 6-11.

Listing 6-10: C code for a nested if statement

```
00401006        mov     [ebp+var_4]  , 0
0040100D        mov     [ebp+var_8]  , 1
00401014        mov     [ebp+var_C], 2
0040101B        mov     eax, [ebp+var_4]
0040101E        cmp     eax, [ebp+var_8]
00401021        jnz     short loc_401047 ❶
00401023        cmp     [ebp+var_C], 0
00401027        jnz     short loc_401038 ❷
00401029        push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E        call    printf
00401033        add     esp, 4
00401036        jmp     short loc_401045
00401038 loc_401038:
00401038        push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D        call    printf
00401042        add     esp, 4
00401045 loc_401045:
00401045        jmp     short loc_401069
00401047 loc_401047:
00401047        cmp     [ebp+var_C], 0
0040104B        jnz     short loc_40105C ❸
0040104D        push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052        call    printf
00401057        add     esp, 4
0040105A        jmp     short loc_401069
0040105C loc_40105C:
0040105C        push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061        call    printf00401061
```

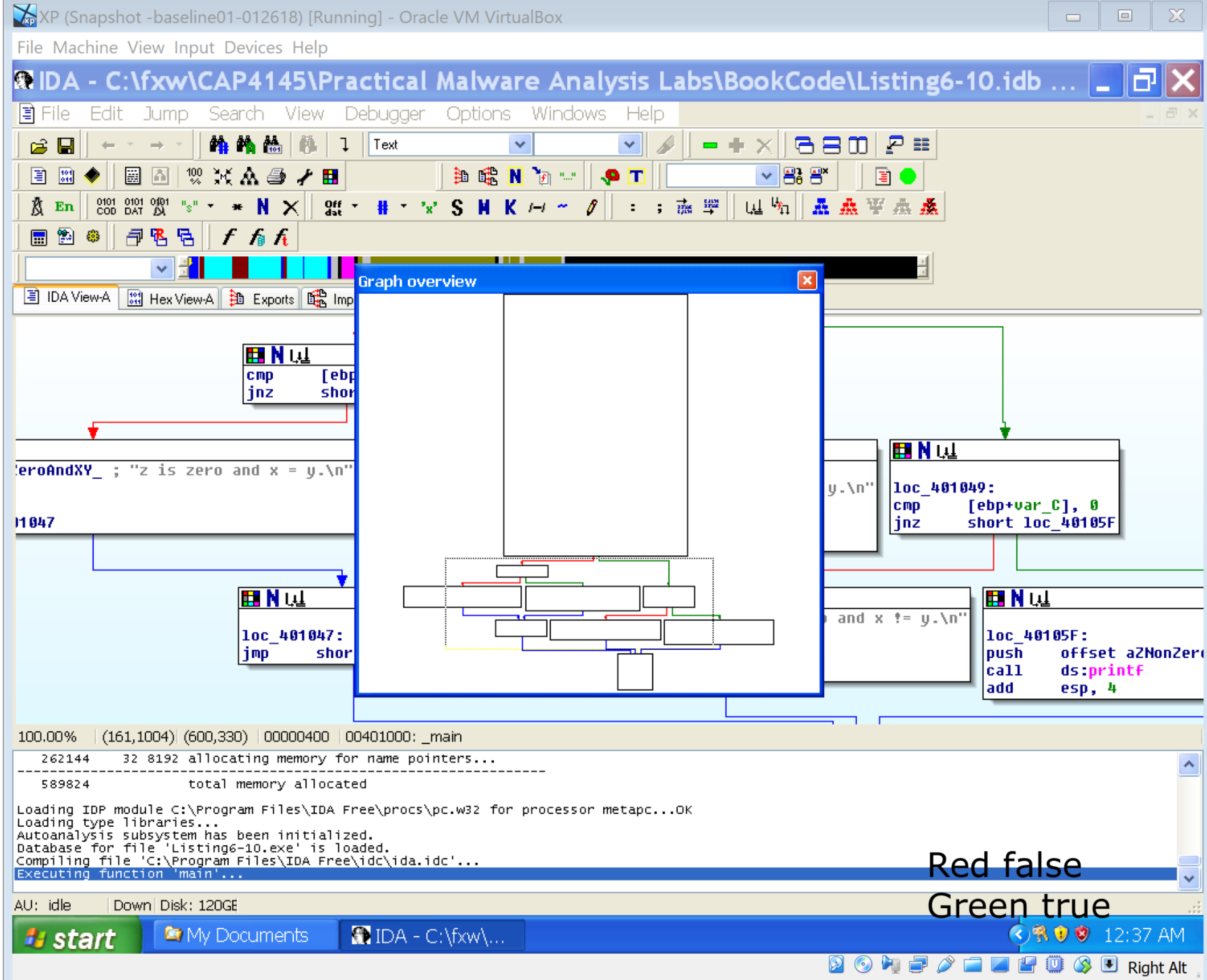
Listing 6-11: Assembly code for the nested if *statement example shown in Listing 6-10*

As you can see, three different conditional jumps occur.

- The first occurs if **var_4** does not equal **var_8** at ①.
- Other two occur if **var_C** is not equal to **zero** at ② and ③.

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main          proc near             ; CODE XREF:
___tmainCRTStartup+10Ap
.text:00401000
.text:00401000 var_4          = dword ptr -0Ch ; int x
.text:00401000 var_8          = dword ptr -8 ; int y
.text:00401000 var_C          = dword ptr -4 ; int z
.text:00401000 argc           = dword ptr  8
.text:00401000 argv           = dword ptr  0Ch
.text:00401000 envp           = dword ptr  10h
.text:00401000
.text:00401000                push    ebp
.text:00401001                mov     ebp, esp
.text:00401003                sub     esp, 0Ch
.text:00401006                mov     [ebp+var_4], 0
.text:0040100D                mov     [ebp+var_8], 1
.text:00401014                mov     [ebp+var_C], 2
.text:0040101B                mov     eax, [ebp+var_4]
.text:0040101E                cmp     eax, [ebp+var_8]
.text:00401021                jnz     short loc_401049
.text:00401023                cmp     [ebp+var_C], 0
.text:00401027                jnz     short loc_401039
.text:00401029                push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
.text:0040102E                call    ds:printf
.text:00401034                add     esp, 4
.text:00401037                jmp     short loc_401047
.text:00401039 ; ---------------------------------------------------------------
```

```
.text:00401039
.text:00401039 loc_401039:                             ; CODE XREF: _main+27j
.text:00401039                 push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
.text:0040103E                 call    ds:printf
.text:00401044                 add     esp, 4
.text:00401047
.text:00401047 loc_401047:                             ; CODE XREF: _main+37j
.text:00401047                 jmp     short loc_40106D
.text:00401049 ; ---------------------------------------------------------------------------
.text:00401049
.text:00401049 loc_401049:                             ; CODE XREF: _main+21j
.text:00401049                 cmp     [ebp+var_C], 0
.text:0040104D                 jnz     short loc_40105F
.text:0040104F                 push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
.text:00401054                 call    ds:printf
.text:0040105A                 add     esp, 4
.text:0040105D                 jmp     short loc_40106D
.text:0040105F ; ---------------------------------------------------------------------------
.text:0040105F
.text:0040105F loc_40105F:                             ; CODE XREF: _main+4Dj
.text:0040105F                 push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
.text:00401064                 call    ds:printf
.text:0040106A                 add     esp, 4
.text:0040106D
.text:0040106D loc_40106D:                             ; CODE XREF: _main:loc_401047j
.text:0040106D                                         ; _main+5Dj
.text:0040106D                 xor     eax, eax
.text:0040106F                 mov     esp, ebp
.text:00401071                 pop     ebp
.text:00401072                 retn
.text:00401072 _main           endp
```



Red false

Green true

Recognizing Loops

- Loops and repetitive tasks are very common in all software and important to be recognized.
- **Finding "for" Loops**
  - Basic looping mechanism used in C programming.
  - Have four components: initialization, comparison, execution instructions, and the "increment or decrement".
- **Finding "while" Loops**
  - "while" loop is frequently used by malware authors to loop until a condition is met, such as receiving a packet or command.
  - "while" loops look like "for" loops in assembly, but they are easier to understand.

Recognizing “for” Loops

```
main() {
    int i;

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

Listing 6-12: C code for a for loop

Four components:

- **Initialization**: i starts at 0
- **Comparison**: is i<100 ?
- **Execution**: printf
- **Increment/decrement**: i++

Recognizing "for" Loops

In this example:

- The initialization sets *i* to 0 (zero), and
- The comparison checks if *i* is less than 100: if yes, then:
  - *The printf instruction will execute, and*
  - *The increment will add 1 to **i**, and*
  - *The process will check to see if **i** is less than 100.*
- These steps will repeat until *i* is greater than or equal to 100.
- How about assembly?
  - *Next slides.*

```
int i;

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

Listing 6-12: C code for a for loop

```
00401004         mov     [ebp+var_4], 0 ❶              Initialization
0040100B         jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D         mov     eax, [ebp+var_4] ❸
00401010         add     eax, 1                        Increment
00401013         mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016         cmp     [ebp+var_4], 64h ❺
0040101A         jge     short loc_40102F ❻            Comparison
0040101C         mov     ecx, [ebp+var_4]
0040101F         push    ecx
00401020         push    offset aID  ; "i equals %d\n"
00401025         call    printf
0040102A         add     esp, 8
0040102D         jmp     short loc_40100D ❼            Execution
```

Listing 6-13: Assembly code for the for loop example in Listing 6-12

Recognizing “for” Loops

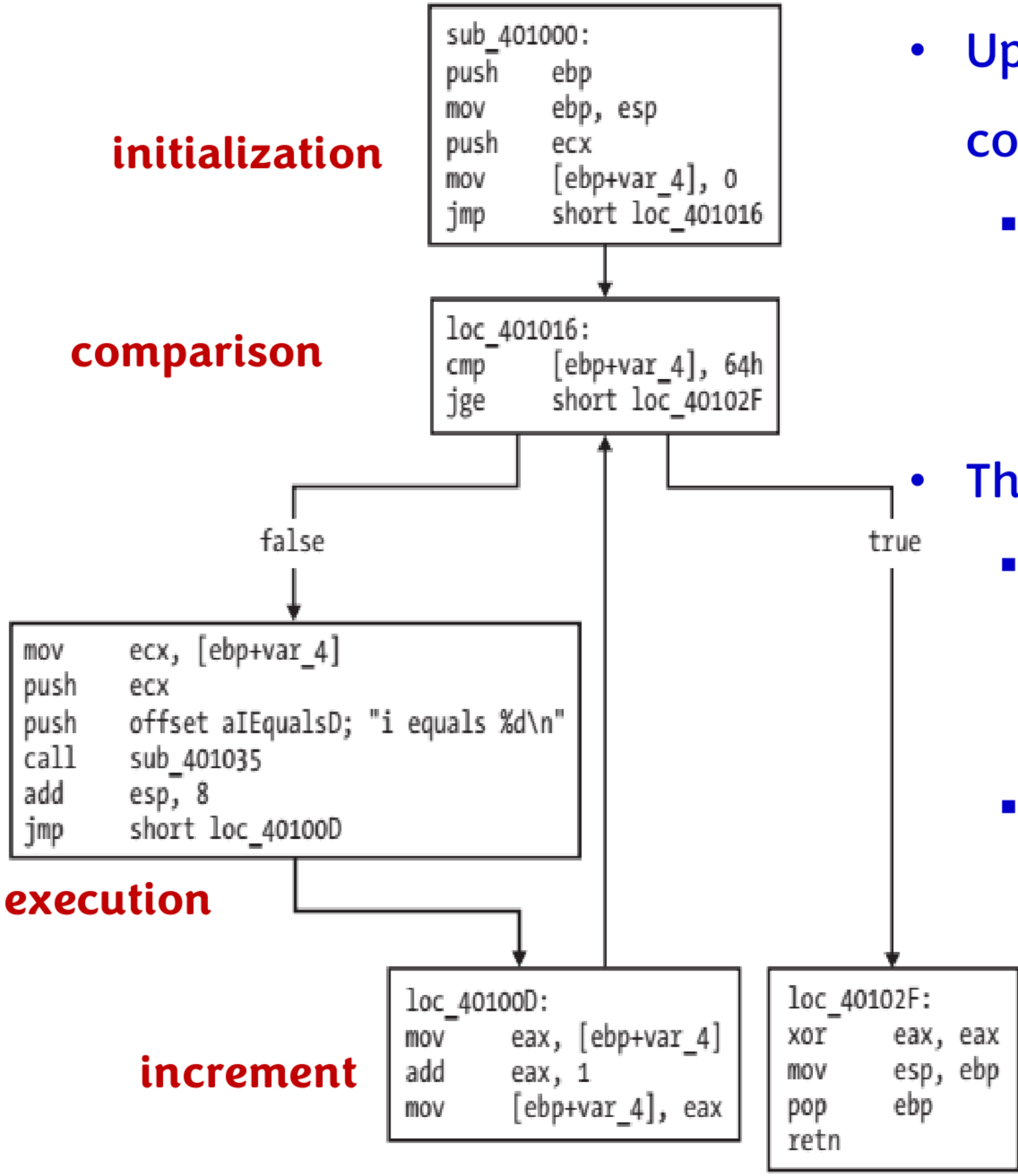
In assembly, the “for” loop can be recognized by locating its four components (INIT/COMP/EXEC/INC-DEC).

For example, in Listing 6-13:

- Code at ① corresponds to the initialization step.
- Code between ③ and ④ corresponds to the increment that is initially jumped over at ② with a jump instruction.
- The comparison occurs at ⑤ and at ⑥; the decision is made by the conditional jump.
- If the jump is not taken, the *printf instruction* will execute, and an unconditional jump occurs at ⑦ which causes the increment to occur.

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main         proc near            ; CODE XREF: ___tmainCRTStartup+10Ap
.text:00401000
.text:00401000 var_4         = dword ptr -4 ; int i
.text:00401000 argc          = dword ptr  8
.text:00401000 argv          = dword ptr  0Ch
.text:00401000 envp          = dword ptr  10h
.text:00401000
.text:00401000               push    ebp
.text:00401001               mov     ebp, esp
.text:00401003               push    ecx
.text:00401004               mov     [ebp+var_4], 0
.text:0040100B               jmp     short loc_401016
.text:0040100D ; ---------------------------------------------------------------------------
.text:0040100D
.text:0040100D loc_40100D:                         ; CODE XREF: _main+2Ej
.text:0040100D               mov     eax, [ebp+var_4]
.text:00401010               add     eax, 1
.text:00401013               mov     [ebp+var_4], eax
.text:00401016
.text:00401016 loc_401016:                         ; CODE XREF: _main+Bj
.text:00401016               cmp     [ebp+var_4], 64h
.text:0040101A               jge     short loc_401030
.text:0040101C               mov     ecx, [ebp+var_4]
.text:0040101F               push    ecx
.text:00401020               push    offset aIEqualsD ; "i equals %d\n"
.text:00401025               call    ds:printf
.text:0040102B               add     esp, 8
.text:0040102E               jmp     short loc_40100D
.text:00401030 ; ---------------------------------------------------------------------------
.text:00401030
.text:00401030 loc_401030:                         ; CODE XREF: _main+1Aj
.text:00401030               xor     eax, eax
.text:00401032               mov     esp, ebp
.text:00401034               pop     ebp
.text:00401035               retn
.text:00401035 _main         endp
```

"for" loop using IDA Pro's graphing.



- Upward arrow after the increment code indicates a loop.
  - Arrows make loops easier to recognize in the graph view.
- The graph displays five boxes:
  - initialization, comparison, execution, increment, and the function epilogue.
  - Epilogue is the portion of a function responsible for cleaning up the stack and returning.

Figure 6-2: Disassembly graph for the for loop example in Listing 6-13

Recognizing "while" Loops

```
int status=0;
int result = 0;

while(status == 0){
        result = performAction();
        status = checkResult(result);
}
```

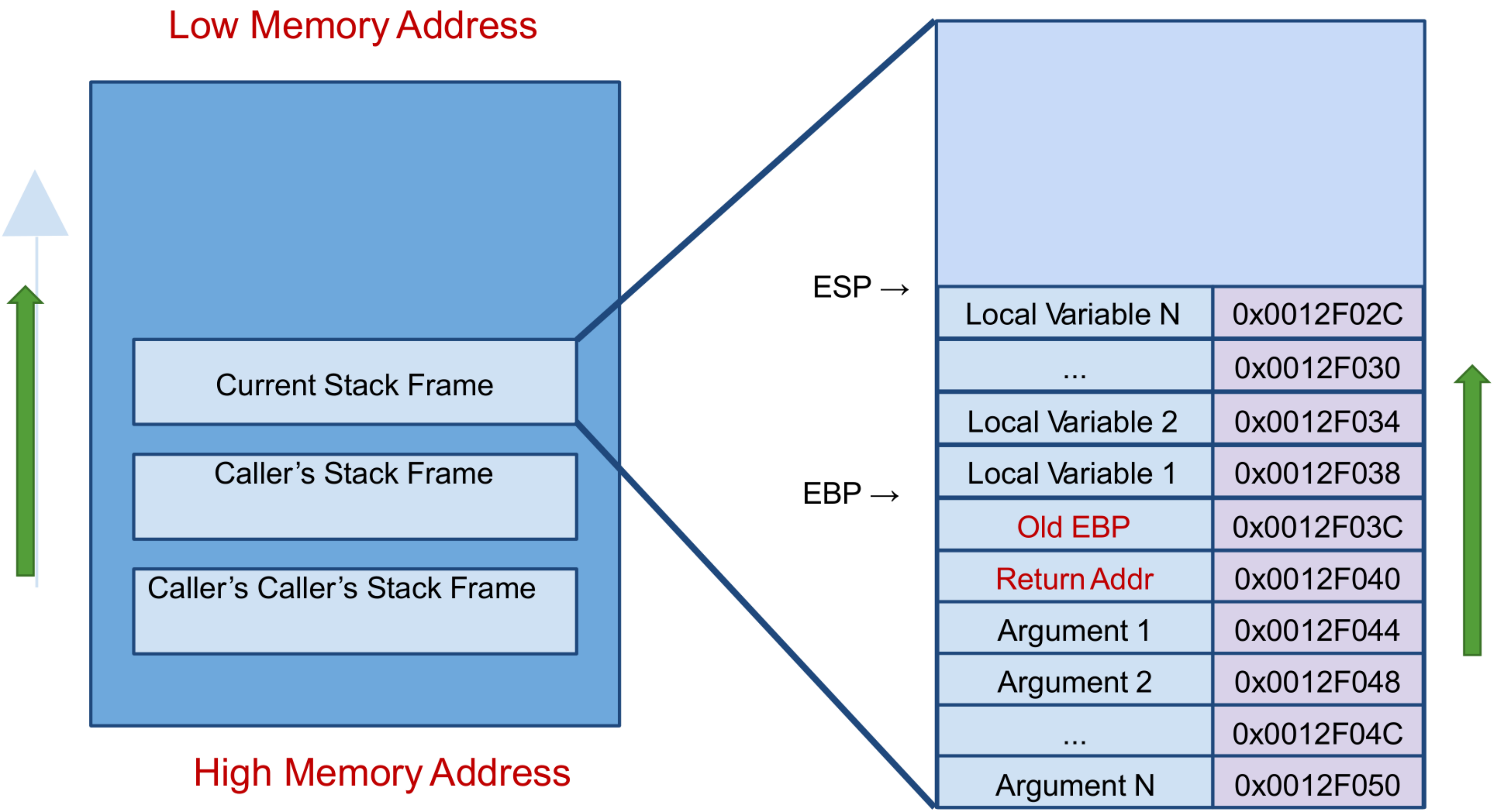
Listing 6-14: C code for a while loop

```
00401036         mov     [ebp+var_4], 0
0040103D         mov     [ebp+var_8], 0
00401044 loc_401044:
00401044         cmp     [ebp+var_4], 0
00401048         jnz     short loc_401063 ❶
0040104A         call    performAction
0040104F         mov     [ebp+var_8], eax
00401052         mov     eax, [ebp+var_8]
00401055         push    eax
00401056         call    checkResult
0040105B         add     esp, 4
0040105E         mov     [ebp+var_4], eax
00401061         jmp     short loc_401044 ❷
```

Listing 6-15: Assembly code for the `while` loop example in Listing 6-14

The assembly code in looks like the for loop, *except that it lacks an increment section*. A conditional jump occurs at ① and an unconditional jump at ②, but the only way for this code to stop executing repeatedly is for that conditional jump to occur.

Understanding Function Call Conventions



Understanding Function Call Conventions

For example: In cdecl,

- *Parameters are pushed onto the stack from right to left,*
- *Caller cleans up the stack when the function is complete,*
- *And the return value is stored in EAX.*

Listing 6-16:

Pseudocode for a function call

```
int test(int x, int y, int z);
int a, b, c, ret;

ret = test(a, b, c);
```

Listing 6-17:

cdecl function call

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

Disassembling Arrays

- *Arrays are* used to define an ordered set of similar data items.
- Malware sometimes uses an array of pointers to strings that contain multiple hostnames that are used as options for connections.
- Listing 6-24 shows two arrays used by one program,
  - Array a is locally defined,
  - Array b is globally defined.
  - These definitions will impact the assembly code.
  - both of which are set during the iteration through the for a loop.

```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

Listing 6-24: C code for an array

Disassembling Arrays

```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

Listing 6-24: C code for an array

```
00401006         mov     [ebp+var_18], 0
0040100D         jmp     short loc_401018
0040100F loc_40100F:
0040100F         mov     eax, [ebp+var_18]
00401012         add     eax, 1
00401015         mov     [ebp+var_18], eax
00401018 loc_401018:
00401018         cmp     [ebp+var_18], 5
0040101C         jge     short loc_401037
0040101E         mov     ecx, [ebp+var_18]
00401021         mov     edx, [ebp+var_18]
00401024         mov     [ebp+ecx*4+var_14], edx ❶
00401028         mov     eax, [ebp+var_18]
0040102B         mov     ecx, [ebp+var_18]
0040102E         mov     dword_40A000[ecx*4], eax ❷
00401035         jmp     short loc_40100F
```

Listing 6-25: Assembly code for the array in Listing 6-24

```
00401006         mov     [ebp+var_18], 0
0040100D         jmp     short loc_401018
0040100F loc_40100F:
0040100F         mov     eax, [ebp+var_18]
00401012         add     eax, 1
00401015         mov     [ebp+var_18], eax
00401018 loc_401018:
00401018         cmp     [ebp+var_18], 5
0040101C         jge     short loc_401037
0040101E         mov     ecx, [ebp+var_18]
00401021         mov     edx, [ebp+var_18]
00401024         mov     [ebp+ecx*4+var_14], edx 1
00401028         mov     eax, [ebp+var_18]
0040102B         mov     ecx, [ebp+var_18]
0040102E         mov     dword_40A000[ecx*4], eax 2
00401035         jmp     short loc_40100F
```

Initialization

Increment

Comparison

Assign value to Element in b (base is var_14)

Assign value to Element in a (base is dword_40A000)

- In assembly, arrays are accessed using a base address as a starting point.
- The size of each element is not always obvious, but it can be determined by seeing how the array is being indexed.
- Listing 6-25 shows the assembly code for Listing 6-24.
  - Base address of array **b** corresponds to **dword_40A000**, and the base address of array **a** corresponds to **var_14**.
  - Since these are both arrays of integers, each element is of **size 4**,
  - Though instructions at ① & ② differ for accessing the two arrays.
    - Both uses **ecx** as an index, which is **multiplied by 4** to account for the size of the elements.
    - The resulting value is added to the base address of the array to access the proper array element.

```
.text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main        proc near           ; CODE XREF: ___tmainCRTStartup+10Ap
.text:00401000
.text:00401000 var_18       = dword ptr -18h ; int i
.text:00401000 var_14       = dword ptr -14h ; int a[5]
.text:00401000 argc         = dword ptr  8
.text:00401000 argv         = dword ptr  0Ch
.text:00401000 envp         = dword ptr  10h
.text:00401000
.text:00401000          push   ebp
.text:00401001          mov    ebp, esp
.text:00401003          sub    esp, 18h ; 18h=24
.text:00401006          mov    [ebp+var_18], 0 ; i=0
.text:0040100D          jmp    short loc_401018
.text:0040100F ; ---------------------------------------------------------------
.text:0040100F
.text:0040100F loc_40100F:                      ; CODE XREF: _main+35j
.text:0040100F          mov    eax, [ebp+var_18]
.text:00401012          add    eax, 1
.text:00401015          mov    [ebp+var_18], eax
.text:00401018
.text:00401018 loc_401018:                      ; CODE XREF: _main+Dj
.text:00401018          cmp    [ebp+var_18], 5
.text:0040101C          jge    short loc_401037
.text:0040101E          mov    ecx, [ebp+var_18]
.text:00401021          mov    edx, [ebp+var_18]
.text:00401024          mov    [ebp+ecx*4+var_14], edx
.text:00401028          mov    eax, [ebp+var_18]
.text:0040102B          mov    ecx, [ebp+var_18]
.text:0040102E          mov    dword_403018[eax*4], ecx ; int b[5]
.text:00401035          jmp    short loc_40100F
.text:00401037 loc_401037:                      ; CODE XREF: _main+1Cj
.text:00401037          xor    eax, eax
.text:00401039          mov    esp, ebp
.text:0040103B          pop    ebp
.text:0040103C          retn
.text:0040103C _main        endp
```

Summary

- **Finding the Code**
  - Strings, then XREF
- **Function Call**
  - Arguments pushed onto the stack
  - Reverse order
  - call
- **Variables**
  - Global: in memory, available to all functions
  - Local: on the stack, only available to one function.

Summary

- **Arithmetic**
  - Move variables into registers
  - Perform arithmetic (add, sub, idiv, etc.)
  - Move results back into variables

- **Branching**
  - Compare (cmp, test, etc.)
  - Conditional jump (jz, jnz, etc.)
  - Red arrow if false, green arrow if true

Main Sources for these slides

- *Michael Sikorski and Andrew Honig, "Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software"; ISBN-10: 1593272901.*
- *Xinwen Fu, "Introduction to Malware Analysis," University of Central Florida*
- *Sam Bowne, "Practical Malware Analysis," City College San Francisco*
- *Abhijit Mohanta and Anoop Saldanha, "Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware," ISBN: 1484261925.*

Thank you