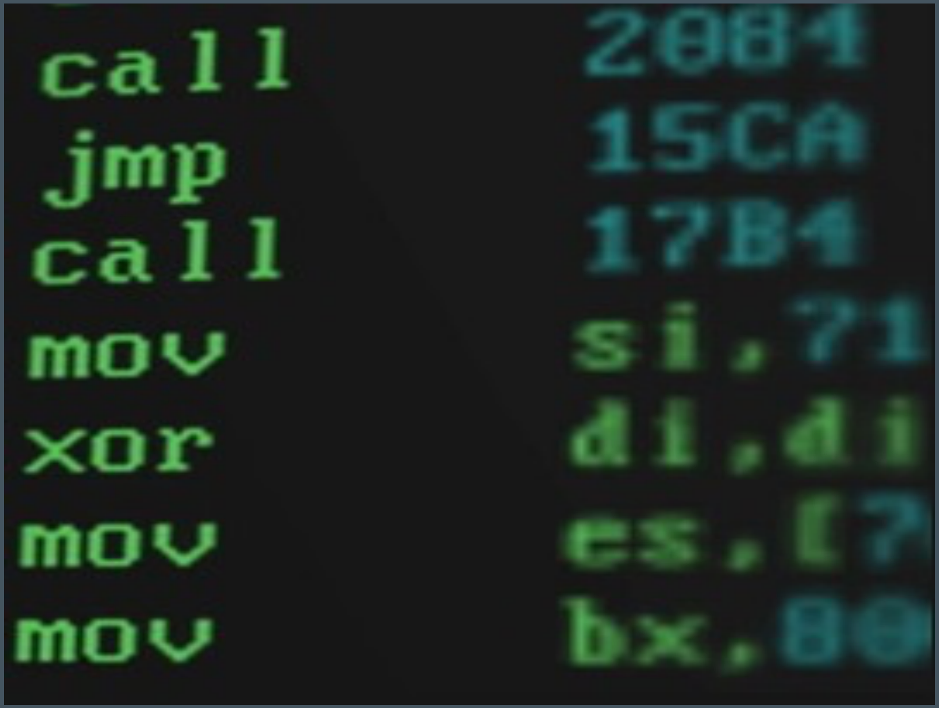
OXFORD BROOKES UNIVERSITY

# COMP6016 Assembly Language Programming III




Dr MUHAMMAD HILMI KAMARUDIN

# Arrays

- Arrays are implemented as fixed size memory buffers
- It is up to the programmer to manage the size of the array elements and to deal with iteration through the array
- Use [] to specify that something is an address, not a value. Some (limited) arithmetic can be done on this.
- Care ***MUST*** be taken over what is a value and what is an address (think pointer arithmetic in C)

# Arrays - addressing

```
mov eax, [ebx]        ; Move the 4 bytes in memory at the
address contained in EBX into EAX
mov [var], ebx        ; Move the contents of EBX into the 4
bytes at memory address var.
                      ; (Note, var is a 32-bit constant).
mov eax, [esi-4]      ; Move 4 bytes at memory address ESI + (-
4) into EAX
mov [esi+eax], cl     ; Move the contents of CL into the byte
at address ESI+EAX
mov edx, [esi+4*ebx]        ; Move the 4 bytes of data at
address ESI+4*EBX into EDX

; The following address calculations are FORBIDDEN!
mov eax, [ebx-ecx]                 ; Can only add register
values
mov [eax+esi+edi], ebx      ; At most 2 registers in address
computation
```

# Arrays – addressing - sizes

- Most of the time, the assembler can work out the memory to use when we move to a memory address but this may not always be possible, especially with constants -

```
mov [rax], 6  ; how long is the 6? 1,2,4, or 8 bytes?
```

- If there is any ambiguity, we need to specify the size of the destination -

```
mov BYTE  [rax], 2   ; Move 2 into the single byte at the address stored in RAX.
mov WORD  [rax], 2   ; Move the 16-bit integer representation of 2 into the 2
                                 ; bytes starting at the address in RAX.
mov DWORD  [rax], 2        ; Move the 32-bit integer representation of 2 into the
                                       ; 4 bytes starting at the address in RAX.
mov QWORD  [rax], 2        ; Move the 64-bit integer representation of 2 into the
                                       ; 8 bytes starting at the address in RAX.
```

# Arrays – times and lea

- Two useful instructions for use with arrays are `times` and `lea`
- times – repeats data or instructions the specified number of times (NB – will not work with macros)

```
zerobuf:         times 64 db 0
buffer: db       'hello, world'
        times 64-$+buffer db ' '
```

- lea – load effective address – put the address specified by the second operand into the register specified by the first operand
- lea parameters are in the form base register, index register, scalar multiplier (must be 1,2,4, or 8), and offset.

```
lea rdx, [rax+ rbx*4+67]
```

# Arrays – a simple example (libasm)

```
; Test programme for assembler arrays
; nasm -g -I libasm_io-master/include -f
elf64 array2.asm
; gcc -g array2.o -lasm_io -o array2

%include
"/home/malware/asm/joey_lib_io_v6_releas
e.asm”

global main
```

# Arrays – a simple example (libasm)

```
section .data

      echo_welcome: db     "Hello, this is a
                           character storage
                           programme",0
      echo_bye: db         "Goodbye!",0
      characters:      times 26 db 0   ; an array
                                         big enough
                                       for the letters
```

# Arrays – a simple example (libasm)

```
section .text

     ; This is our procedure for putting the characters into the array
     populate:
          lea rax, [characters]
          loop1:
               mov [rax],rbx
               inc rbx
               inc rax
               cmp rbx, 91
               jne loop1
          ret
```

# Arrays – a simple example (libasm)

```
; This prints them out
    display:
        loop2:
            movzx rdi, BYTE
[characters+rbx]
            call print_char_new
                    ;PRINT_CHAR rdi
                    ;NEWLINE
            call print_nl_new
            inc rbx
            cmp rbx, 26
            jne loop2
        ret
```

# Arrays – a simple example (libasm)

```
main:
    mov rbp, rsp; for correct debugging
                push rbp
                mov rbp, rsp
                sub rsp,32

                mov rdi, QWORD echo_welcome
                     call print_string_new
                     ;PRINT_STRING [rdi]
                     call print_nl_new
                     ;NEWLINE
                mov rbx, 65    ; ASCII character for A
                call populate
                xor rbx,rbx
                call display
```

# Arrays – a simple example (libasm)

```
mov rdi, QWORD echo_bye
call print_string_new    ;PRINT_STRING [rdi]
call print_nl_new  ;NEWLINE

add rsp, 32

pop rbp

ret
```

# Arrays – a more complex example

```
; Test programme for assembler arrays
; nasm -g -I libasm_io-master/include -f
elf64 array3.asm
; gcc -g array3.o -lasm_io -o array3

%include
"/home/malware/asm/joey_lib_io_v6_release.asm
"
global main
```

# Arrays – a more complex example

```
section .data


echo_welcome: db  "Hello, this is a student ID storage programme",0

echo_instruction: db "Please enter a student ID: ",0

echo_message: db "A valid ID is :",0

echo_bye:   db    "Goodbye!",0

counter:            dq    0      ; a loop counter
characters: times 90 db 0      ; an array big enough for 10 student IDs (8 chars + null)
```

# Arrays – a more complex example

```
section .text

; This is our procedure for reading the IDs
populate:
     lea rbx, [characters]
     mov QWORD  [counter],  0
     loop1:
          mov rdi, QWORD echo_instruction
               call print_string_new
               call print_nl_new
               call read_string_new
```

# Arrays – a more complex example

```
; We can't move a memory address to another
memory address using mov so we use rcx to
store the address

mov rcx,[rax]
mov [rbx], rcx
add rbx,9
add QWORD [counter],9
cmp QWORD [counter], 27
jne loop1
ret
```

# Arrays – a more complex example

```
; This prints them out
      display:
         loop2:
            mov rdi, QWORD echo_message
            call print_string_new ;    set rdi to be
the start of the buffer
            mov rdi,  characters    ; and add the
number of bytes to get to the current record
            add rdi, rbx
            call print_string_new
            call print_nl_new
            add rbx,9
            cmp rbx, 27
            jne loop2

 ret
```

# Arrays – a more complex example

```
main:
    mov rbp, rsp; for correct debugging
        push rbp
        mov rbp, rsp
        sub rsp,32

        mov rdi, QWORD echo_welcome
        call print_string_new
        call print_nl_new

        call populate
        sub rbx,rbx
        call display
```

# Arrays – a more complex example

```
mov rdi, QWORD echo_bye
call print_string_new
call print_nl_new

add rsp, 32

pop rbp

ret
```

# Strings

- Strings are arrays of bytes, however there are instructions to help us more efficiently
- Need to consider if we have sentinel style strings (like C) where there is a known character at the end (normally NULL)
- Or fixed length strings (like Pascal) where the length is encoded as part of the string
- For assembler, most of the time we prefer sentinel strings, but fixed length have their uses
- Also need to be away that many of the strings that we use are essentially byte blocks and the string optimisations are applicable elsewhere

# Strings – length of a constant string

```
; Test programme for assembler arrays
; nasm -g -I libasm_io-master/include -f elf64 strings.asm
; gcc -g strings1.o -lasm_io -o strings

%include "/home/malware/asm/joey_lib_io_v6_release.asm"

global main

section .data

        echo_welcome: db    "Hello, this is a long message",0
        len    equ    $ - echo_welcome
        echo_message: db "It is this many characters :",0
        echo_bye:     db     "Goodbye!",0
```

# Strings – length of a constant string

```
section .text

        main:
                push rbp
                mov rbp, rsp
                sub rsp,32

                mov rdi, QWORD echo_welcome
                call print_string_new
                call print_nl_new

                mov rdi, QWORD echo_message
                call print_string_new
                mov rdi, len
                dec rdi
                call print_uint_new
                call print_nl_new

                mov rdi, QWORD echo_bye
                call print_string_new
                call print_nl_new

                add rsp, 32

                pop rbp

                ret
```

# Strings – stosb & movsb

- movs(b/w/d/q) – copy the (byte/word/double word/quad word) from memory address stored in rsi to memory address stored in rdi
- stos(b/w/d/q) – copies the data from (AL/AX/EAX/RAX) to memory address stored in rdi
- We use movs? to copy from memory location to memory location. This is used to copy blocks of memory
- We use stos? to copy from a register to a memory location – this is used to initialise an area of memory

# Strings – rep

- The instruction rep is used to allow us to repeat stos? and movs? multiple times, enabling fast memory copies and initialisation.
- To use rep, we need to specify -
- The direction of the transfer (high to low (std) or low to high (cld))
- For stos?, the value to copy in which is stored in AL, AX, EAX, or RAX
- For movs?, the location to copy from which is stored in ESI or RDI
- The location to copy to which is stored in EDI or RDI
- The number of bytes to copy which is stored in CL, CX, ECX, or RCX

# Strings – length of a constant string

```
; Test programme for assembler arrays
; nasm -g -I libasm_io-master/include -f elf64 strings2.asm
; gcc -g strings1.o -lasm_io -o strings2

%include "/home/malware/asm/joey_lib_io_v6_release.asm"

global main

section .data

      echo_welcome: db    "Hello, this is a long message",0
      len    equ    $ - echo_welcome
      echo_message: db "This is it changed to be all As",0
      echo_bye:     db    "Goodbye!",0
```

# Strings – length of a constant string

```
section .text

main:
        push rbp
        mov rbp, rsp
        sub rsp,32

        mov rdi, QWORD echo_welcome
        call print_string_new
        call print_nl_new

        mov rdi, QWORD echo_message
        call print_string_new
        call print_nl_new

        Cld                    ; set the direction
        mov al, 'A'            ; we want to overwrite with 'A'
        mov edi, echo_welcome; we want to overwrite the
                                   echo_welcome string
        mov ecx, len+1         ; we need to add 1 here as the
                               string starts at 0
        rep stosb              ; repeat
        mov byte [echo_welcome+len+1],0   ; add the null
                                          terminator
```

# Strings – length of a constant string

```
mov rdi, QWORD echo_welcome
call print_string_new
call print_nl_new

mov rdi, QWORD echo_bye
call print_string_new
call print_nl_new

add rsp, 32

pop rbp

ret
```

# Strings – cmps

- We can compare strings using cmps(b/w/d/q) – this will compare the values at the addresses pointed to by (DI, EDI, or RDI) and (SI, ESI, or RSI) respectively
- cmps can be combined with rep, repe (repeat while a condition is zero and CX!=0), or repne (repeat while a condition is not zero or CX!=0) to check blocks of memory

## Strings – scas

- We can search for a character, or set of characters using the instruction scas(b/w/d/q)
- The register (AL, AX, EAX, or RAX) contains the item we want to search for, and the string to search is at the location stored in (DI, EDI, or RDI)
- The register (CX, ECX, or RCX) is used to limit how far we will search.
- When used in conjunction with repne we can scan through a string and when we have found the substring we are looking for, the register (DI, EDI, or RDI) will contain the position in the string of the substring in memory (take off the start of the string to find out how far into the string something is)

# Strings – length of a variable string

```
; Test programme for assembler arrays
; nasm -g -I libasm_io-master/include -f elf64 strings3.asm
; gcc -g strings3.o -lasm_io -o strings3

%include "/usr/local/include/libasm_io.inc"

global main

section .data

       echo_welcome: db    "Please enter a string",0
       echo_message: db    "The string is this long: ",0
       echo_bye:     db    "Goodbye!",0
```

# Strings – length of a variable string

```
section .text
        main:
                push rbp
                mov rbp, rsp
                sub rsp,32

                mov rdi, QWORD echo_welcome
                call print_string
                call print_nl
                call read_string
                mov rbx, rax                                      ; move the address of
the string
                                                                           ; somewhere
safe

                mov rdi, QWORD echo_message
                call print_string

                sub rcx,rcx                               ; this sets the max size to
look for to be
                not rcx                          ; the maximum memory size
```

# Strings – length of a variable string

```
sub al,al                         ; We want to look for the null terminator
mov rdi, rbx                      ; set the start of the string
cld
repne scasb                                ; search
sub rdi, rbx                      ; we need to take off the start of the
string

dec rdi                           ; and allow for the null terminator
call print_uint
call print_nl

mov rdi, QWORD echo_bye
call print_string
call print_nl

add rsp, 32

pop rbp

ret
```

# Understanding C Code construct in Assembly

- Malware analyst need to be able to obtain a high-level picture of the code functionality.
- Skill that need time to develop.
- Typically malware is developed using high-level language, commonly C.
- A *code construct* is a code abstraction level that defines a functional property ( not detail implementation).
- Eg: loops, conditional statement and so on.
- Discuss on popular C code construct.
- Malware analyst need to be able to go from disassembly to high level constructs,
- For help with C language, have a look look at the classic *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1988)
- Remember that the goal is to understand the overall functionality of a program, not to analyze every single instruction.

# Global vs Local Variables

- *Global variables* can be accessed and used by any function in a program.
- *Local variables* can be accessed only by the function in which they are defined. Both global and local variables are declared similarly in C, but they look completely different in assembly.

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

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

*Figure: A simple program with two global variables*

# Global vs Local Variables cont

- *Global variables* can be accessed and used by any function in a program.
- *Local variables* can be accessed only by the function in which they are defined. Both global and local variables are declared similarly in C, but they look completely different in assembly.

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

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

*Figure 1: A simple program with two global variables*

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

   int y = 2;

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

*Figure 2: A simple program with two global variables*

# Global vs Local Variables cont

OXFORD BROOKES UNIVERSITY

```
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
```

*Fig 3: Assembly code for the global variable example in* Fig 1

```
00401006    mov     dword ptr [ebp-4], 0
0040100D    mov     dword ptr [ebp-8], 1
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
```

*Fig 4: Assembly code for the local variable example in Fig 2*

- The global variables are referenced by memory addresses, and the local variables are referenced by the stack addresses.
- In Listing Fig 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.
- In Fig 4, memory location [ebp-4] is used consistently throughout this function to reference the local variable x.

# Disassembling Arithmetic Operations

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

Fig 5: *C code with two variables and a variety of arithmetic*

Figure 5 shows the C code for two variables and a variety of arithmetic operations. Two of these are the -- and ++ operations, which are used to dec- rement by 1 and increment by 1, respectively. The % operation performs the *modulo* between the two variables, which is the remainder after performing a division operation.

OXFORD BROOKES UNIVERSITY

# Disassembling Arithmetic Operations

```
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
```

Fig 6: *Assembly code for the arithmetic example in Fig 5*

- 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 per-forming the div or idiv instruction (5), we 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).

# Recognizing if Statements

- If statement use by programmer to alter program execution based on certain conditions. if statements are common in C code and disassembly.
- Figure 7 displays a simple if statement in C with the assembly for this code shown in Figure 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.

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

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

*Fig 7: C code if statement example*

```
00401006         mov    [ebp+var_8], 1
0040100D         mov    [ebp+var_4], 2
00401014         mov    eax, [ebp+var_8]
00401017         cmp    eax, [ebp+var_4] 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
```

*Fig 8: Assembly code for the if statement example in Fig 7*

# Graphic function with IDA PRO

- IDA Pro has a graphing tool that is useful in recognizing constructs, as shown in Figure 9. This feature is the default view for analyzing functions.

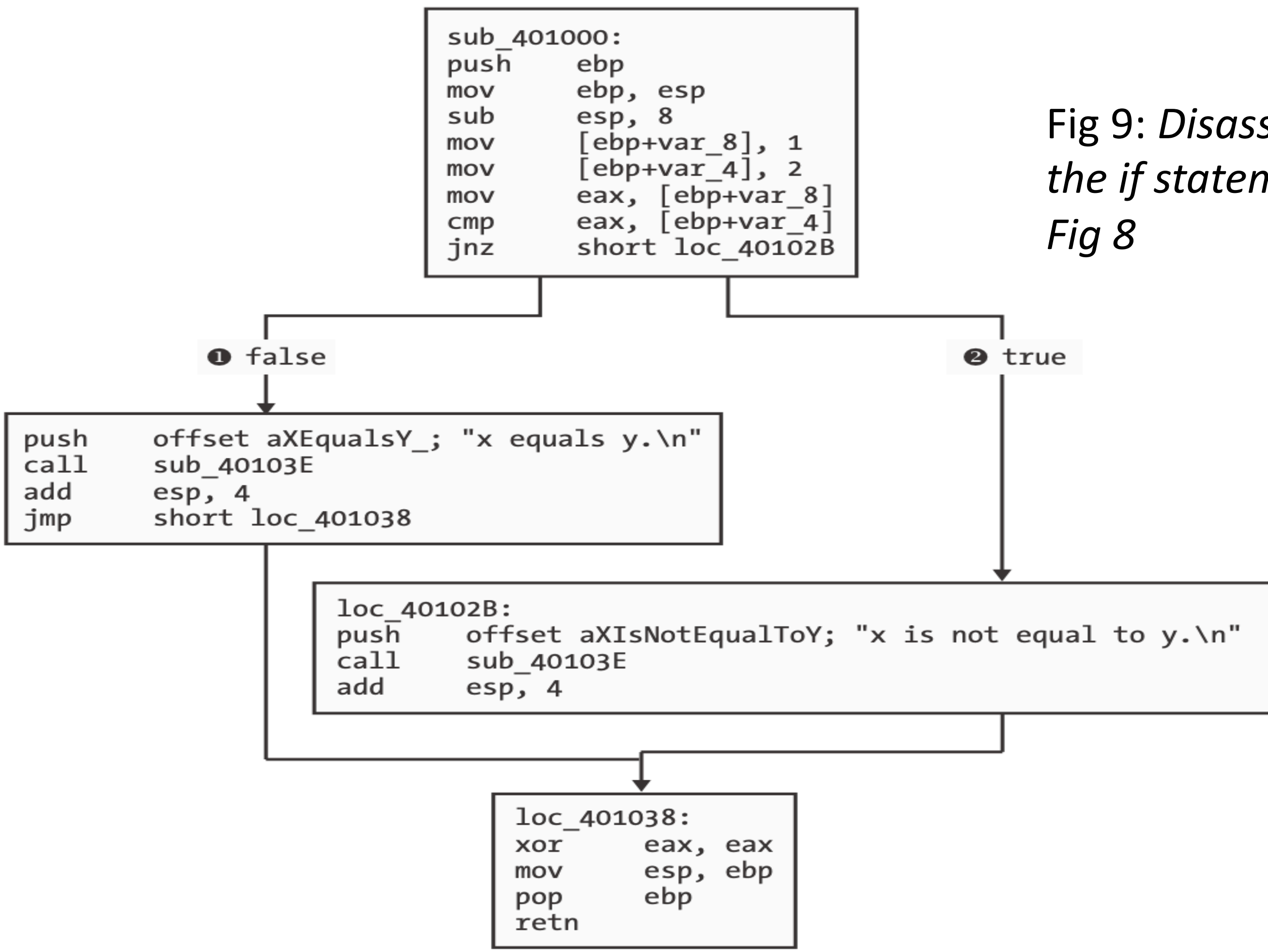



Fig 9: *Disassembly graph for the if statement example in Fig 8*

# Recognizing for Loops

The for loop is a basic looping mechanism used in C programming. for loops always have four components: initialization, comparison, execution instructions, and the increment or decrement.

```
int i;

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

Fig 10: *C code for a for loop*

- The initialization sets i to 0 (zero), and the comparison checks to see if i is less than 100.
- If i is less than 100, the printf instruction will execute, 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

OXFORD BROOKES UNIVERSITY

```
00401004        mov     [ebp+var_4], 0 ❶
0040100B        jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D        mov     eax, [ebp+var_4] ❸
00401010        add     eax, 1
00401013        mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016        cmp     [ebp+var_4], 64h ❺
0040101A        jge     short loc_40102F ❻
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
```

Fig 11: *Assembly code for the for loop example in Fig 10*

- In assembly, the for loop can be recognized by locating the four components—initialization, comparison, execution instructions, and increment/ decrement.
- For example, in Figure above, (1) corresponds to the initialization step. The 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.

# Loops Graph Mode

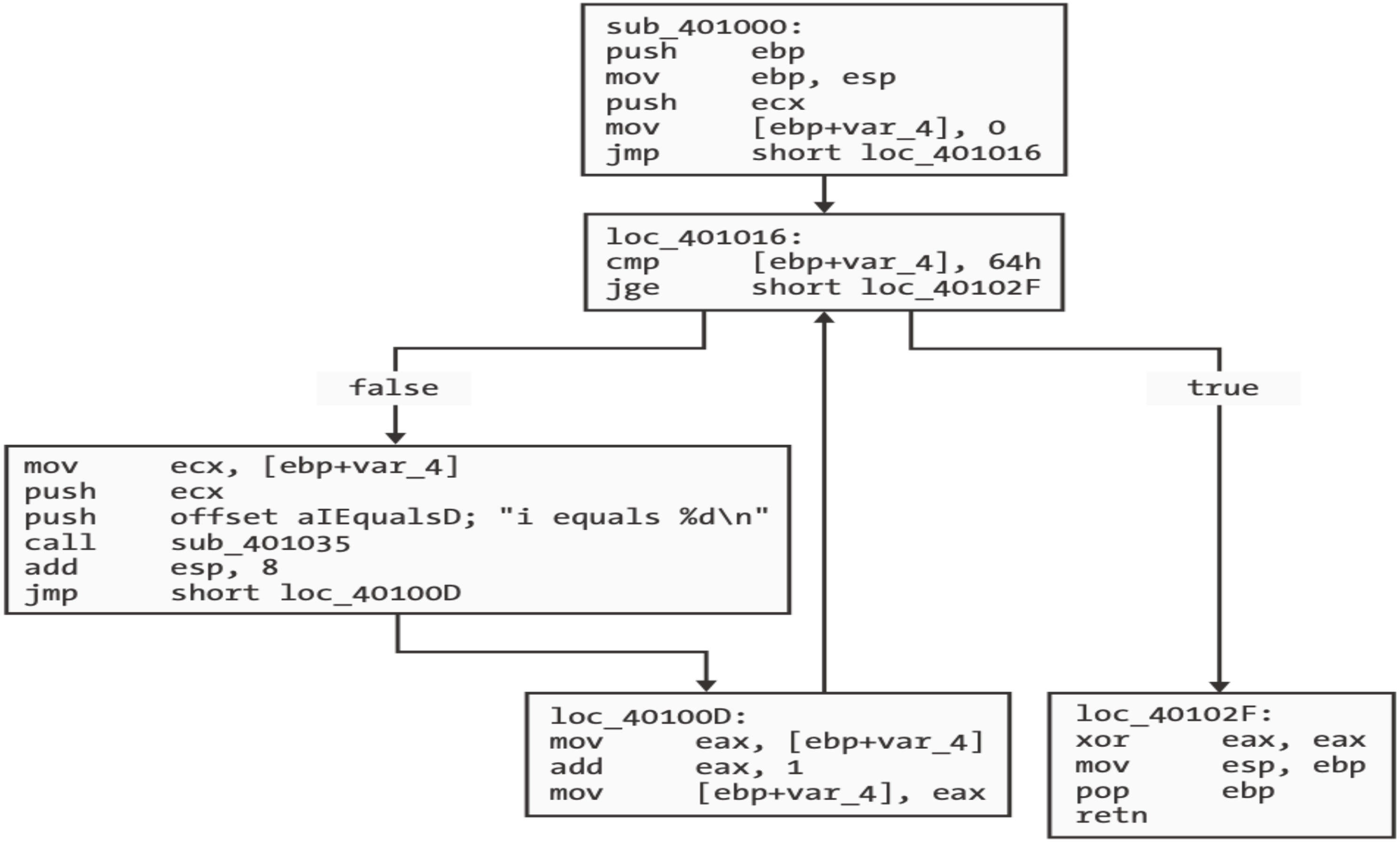



Figure 12: *Disassembly graph for the for loop example in Fig 11*

# Recognizing While Loops

The 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 similar to for loops in assembly, but they are easier to understand.

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

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

Fig 13: *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
```

Fig 14: *Assembly code for the while loop example in Fig 13*

- The while loop in Fig 13 will continue to loop until the status returned from checkResult is 0.
- 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.
- Revision: Read PMA page 119-132