Practical Malware Analysis

Lecture 6 | C Code Constructs in Assembly

Goals

- Efficient analysis
  - Focus on functionality over implementation details
  - Recognize code constructs (if/switch statements, for/while loops, etc)

- Recognize differences between compilers

Goals

Efficient analysis
Focus on functionality over implementation details
Recognize code constructs (if/switch statements, for/while loops, etc)

Recognize differences between compilers

Global vs Local Variables

- Global variables referenced by memory address

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

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

```
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
```

The global variable X is represented by dword_40CF60, a memory location at 0x40CF60. When X is updated, the value at that location in memory is changed, affecting any other reference to that global variable.

Global vs Local Variables

- Local variables referenced by stack address

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

   int y = 2;

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

```
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
```

The local variable X is represented by its constant offset relative to ebp, [ebp-4], useful only to the function in which X is defined.

Arithmetic Operations

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

```
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
```

Local variables *a* and *b* are set to 0 and 1 respectively.
At (1) var_4 (*a)* is set to the result of adding *a* to 0xB (11 in decimal).
At (2) we can see *a* being moved into ecx and then var_8 (*b*) is subtracted from it with the result being stored back into *a*.
At (3) we can see *a* being move into edx and then having 1 subtracted, with the results stored back into *a*. Note that the compiler chose to use the sub instruction instead of the dec instruction.
At (4) a similar procedure occurs with *b*, but using eax and the add instruction instead of the inc instruction.
Above (5) we see *a* loaded into eax and converted into a double word that will span edx and eax. The idiv instruction will divide edx:eax by ecx (set to 3), storing the remainder in edx which is then moved into *b*.

if Statements

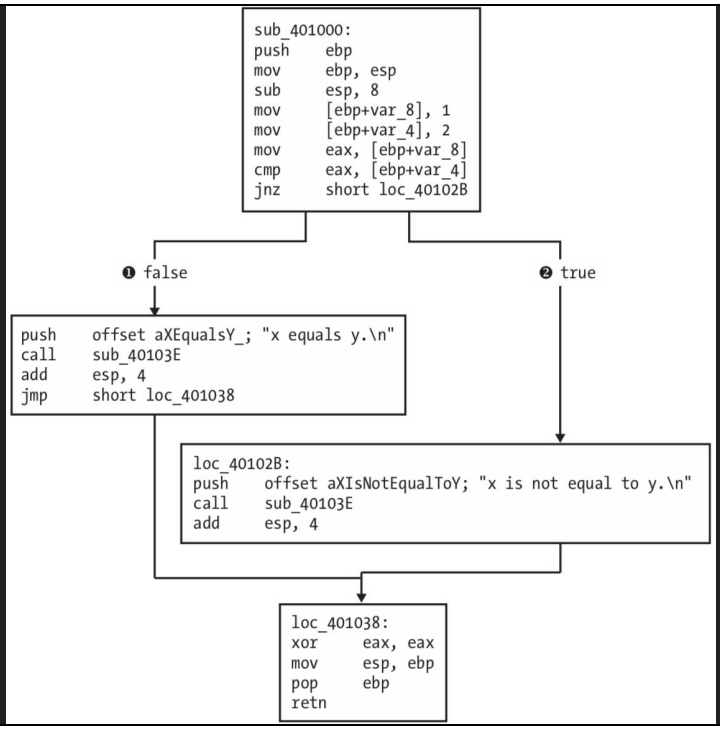
```
int x = 1;
int y = 2;

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

```
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
```

We can see *x* (var_8) being initialized to 1 and *y* (var_4) to 2 before the two values are compared. If the two values are equal (x == y) then the comparison will set the zero flag (ZF) and the conditional jump will not be taken, printing “x equals y.” before adjusting the stack pointer and unconditionally jumping to the next instruction at 0x00401038. Note that if this happens, there is no way for the code in the “else” part of the if statement to execute.

If the two values are not equal, the zero flag will not be set and the jnz instruction will jump execution to 0x0040102B, printing “x is not equal to y.”



This is the graph view representation of the if statement from the previous slide.

Nested if Statements

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

```asm
00401006        mov     [ebp+var_8], 0
0040100D        mov     [ebp+var_4], 1
00401014        mov     [ebp+var_C], 2
0040101B        mov     eax, [ebp+var_8]
0040101E        cmp     eax, [ebp+var_4]
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
```

We can see three different conditional jumps highlighted in the above assembly.

(1) which occurs if var_8 != var_4 (testing for x == y)
(2) which occurs if (1) is not taken (x == y) and var_C != 0 (testing for z == 0)
(3) which occurs if (1) is taken and var_C != 0

for Loops

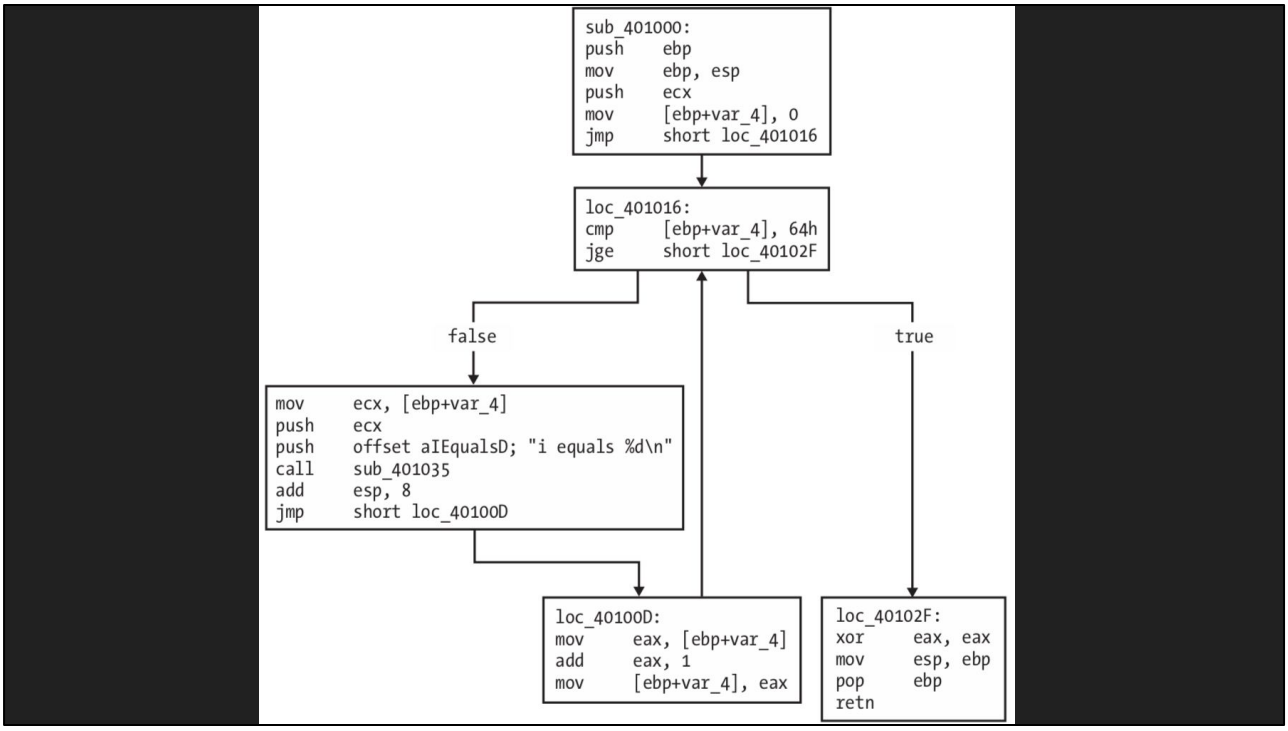
Four components: initialization, comparison, instruction executions, and the increment or decrement

```
int i;

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

```
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
```

For loops are easy to recognize in assembly, as you can pick out where the loop counter is initialized (i = 0 at (1)), where the comparison occurs (i < 100 at (5)), if the comparison is true the instructions after (6) will execute and jump (7) to the increment (i++ at (3, 4)). If the comparison returns false (i >= 100) then the conditional jump at (6) will be taken, exiting the loop. Note that the increment is initially skipped by (2) in order for the loop code to run once prior to the loop counter being adjusted.



As we learned previously, for loops are easily observed in IDA Pro's graph view, as they include an up arrow after the increment/decrement which returns to the comparison to determine if the code inside of the loop should be executed again or not.

The four components of a for loop are easily observed in each of the four above boxes, with the bottom right box showing the function epilogue, which represents the callee cleaning up the stack before returning execution to the caller function.

while Loops

Like for loops without the inc/dec

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

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

```
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
```

Looking at the assembly of the above while loop, it is easy to see that it is much like the previous for loops, but lacking instructions for incrementing or decrementing a loop counter.

Other Calling Conventions

Convention	Parameters	Cleanup
cdecl	Pushed onto stack from right to left	Caller cleans Return value -> eax
stdcall (Windows API)	Pushed onto stack from right to left	Callee cleans
Fastcall (Microsoft)	First few -> registers (ECX & EDX), rest pushed from left to right	Caller cleans

The above table highlights key differences between the three most common calling conventions

Table 6-1: Assembly Code for a Function Call with Two Different Calling Conventions

Visual Studio version			GCC version		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

The above table provides an example of some assembly for the same code compiled by both Visual Studio and GCC. Key differences to take note of are how Visual Studio pushes parameters onto the stack prior to calling the adder function, while GCC moves the parameters onto the stack prior to calling the function. Visual studio also uses an extra function to restore the stack (highlighted in bold) which is not necessary for GCC, as it never altered the stack pointer.

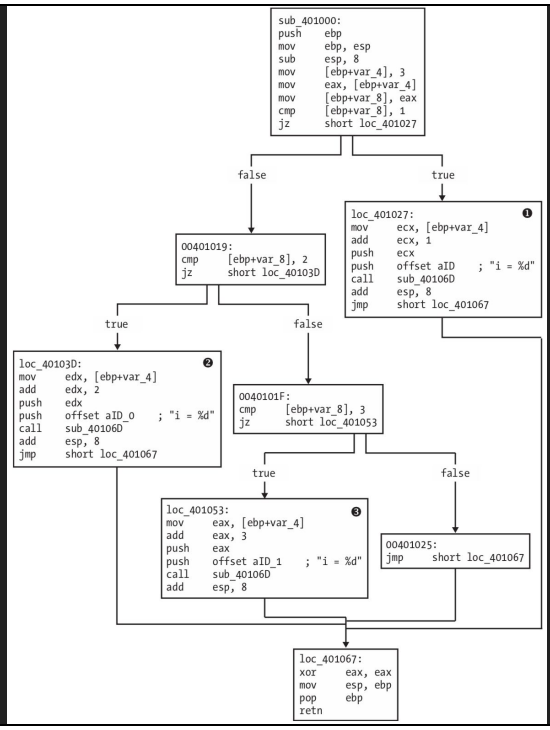
switch Statements - if Style

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

```
00401013          cmp     [ebp+var_8], 1
00401017          jz      short loc_401027 ❶
00401019          cmp     [ebp+var_8], 2
0040101D          jz      short loc_40103D
0040101F          cmp     [ebp+var_8], 3
00401023          jz      short loc_401053
00401025          jmp     short loc_401067 ❷
00401027 loc_401027:
00401027          mov     ecx, [ebp+var_4] ❸
0040102A          add     ecx, 1
0040102D          push    ecx
0040102E          push    offset unk_40C000 ; i = %d
00401033          call    printf
00401038          add     esp, 8
0040103B          jmp     short loc_401067
0040103D loc_40103D:
0040103D          mov     edx, [ebp+var_4] ❹
00401040          add     edx, 2
00401043          push    edx
00401044          push    offset unk_40C004 ; i = %d
00401049          call    printf
0040104E          add     esp, 8
00401051          jmp     short loc_401067
00401053 loc_401053:
00401053          mov     eax, [ebp+var_4] ❺
00401056          add     eax, 3
00401059          push    eax
0040105A          push    offset unk_40C008 ; i = %d
0040105F          call    printf
00401064          add     esp, 8
```

For this simple switch statement, we can see that each case is handled by a pair of cmp and jz instructions between (1) and (2) in the assembly on the right, with a final, unconditional jmp to handle the default case. Each location referenced by each conditional jump contains independendent blocks of instructions, as they each end with an unconditional jmp to the instructions for the default/break case.

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```



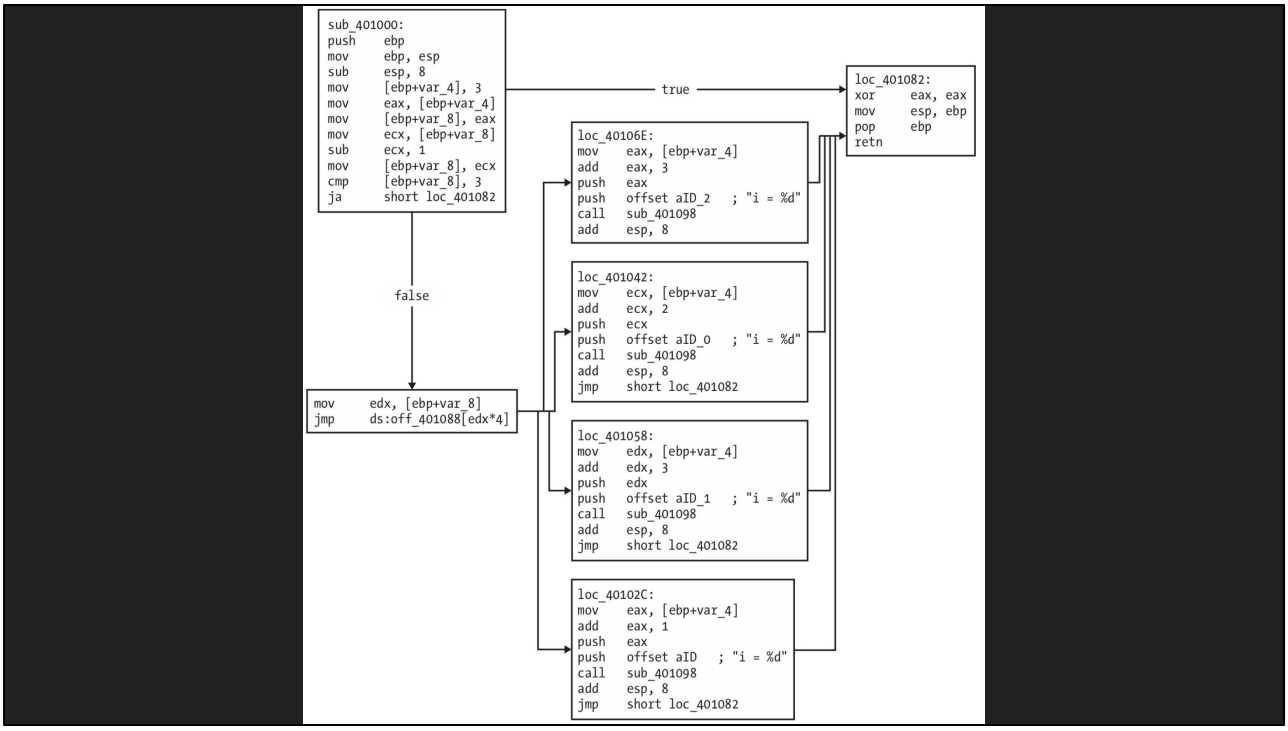
In the graphical view of the switch statement, it is difficult to tell if the original code was a switch statement or a series of if statements, the instructions look the same.

switch Statements - Jump Table

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   case 4:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

```
00401016        sub     ecx, 1
00401019        mov     [ebp+var_8], ecx
0040101C        cmp     [ebp+var_8], 3
00401020        ja      short loc_401082
00401022        mov     edx, [ebp+var_8]
00401025        jmp     ds:off_401088[edx*4] ❶
0040102C   loc_40102C:
              ...
00401040        jmp     short loc_401082
00401042   loc_401042:
              ...
00401056        jmp     short loc_401082
00401058   loc_401058:
              ...
0040106C        jmp     short loc_401082
0040106E   loc_40106E:
              ...
00401082   loc_401082:
00401082        xor     eax, eax
00401084        mov     esp, ebp
00401086        pop     ebp
00401087        retn
00401087   _main   endp
00401088 ❷off_401088  dd offset loc_40102C
0040108C              dd offset loc_401042
00401090              dd offset loc_401058
00401094              dd offset loc_40106E
```

Consider the case when a fourth switch is added to the switch statement. The compiler optimizes the code to avoid performing many unnecessary comparisons using a jump table (2), which uses the switch variable as an index into the table containing offsets to various locations in memory which contain the code to be executed based on the case. In this example, ecx contains the switch variable and is decremented by one in order for the jump table to be properly indexed from 0 to 3 (since the switch range is 1 to 4). The jump instruction at (1) multiplies the switch variable (minus one) with 4 in order to access the proper target offset in the jump table (where each entry is a memory address which is 4 bytes long).



(graph view of switch statement from previous slide)

Arrays

```c
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;
   }
}
```

```
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
```

Observe two arrays, Array *b* is globally defined and array *a* is locally defined. At (1) we can see the value of index variable *i* at [ebp+var_18] being moved into both the ecx and edx registers, with ecx used to access the proper index of array *a* (ecx*4), var_14 representing the base address of the local array *a,* and the value 4 corresponding to the size of each element in an integer array. At (2) the value of *i* is again loaded into ecx and eax, with ecx*4 used to index the array based at address 0x0040A0000.

Structs

- Like untyped arrays
- Commonly used to group info
- Windows API functions

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
           q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

Structures (structs) are similar to arrays, but they may contain elements of varying types. Structures may be used by malware authors to group information, and are used for Windows API functions that require the calling function to create and maintain them.

Structs

- Accessed by base address
- Nearby data types may be coincidental

```
00401050        push    ebp
00401051        mov     ebp, esp
00401053        push    20h
00401055        call    malloc
0040105A        add     esp, 4
0040105D        mov     dword_40EA30, eax
00401062        mov     eax, dword_40EA30
00401067        push    eax ❶
00401068        call    sub_401000
0040106D        add     esp, 4
00401070        xor     eax, eax
00401072        pop     ebp
00401073        retn
```

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
           q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

Structs are accessed similarly to arrays, with a base address (0x0040EA30 since gms is a global variable). At (1) gms is pushed onto the stack as the argument to the test() function at 0x00401000.

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
           q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

```
00401000          push    ebp
00401001          mov     ebp, esp
00401003          push    ecx
00401004          mov     eax,[ebp+arg_0]
00401007          mov     byte ptr [eax+14h], 61h
0040100B          mov     ecx, [ebp+arg_0]
0040100E          fld     ds:dbl_40B120 ❶
00401014          fstp    qword ptr [ecx+18h]
00401017          mov     [ebp+var_4], 0
0040101E          jmp     short loc_401029
00401020 loc_401020:
00401020          mov     edx,[ebp+var_4]
00401023          add     edx, 1
00401026          mov     [ebp+var_4], edx
00401029 loc_401029:
00401029          cmp     [ebp+var_4], 5
0040102D          jge     short loc_40103D
0040102F          mov     eax,[ebp+var_4]
00401032          mov     ecx,[ebp+arg_0]
00401035          mov     edx,[ebp+var_4]
00401038          mov     [ecx+eax*4],edx ❷
0040103B          jmp     short loc_401020
0040103D loc_40103D:
0040103D          mov     esp, ebp
0040103F          pop     ebp
00401040          retn
```

In the test function, we can see the base address of the local structure is arg_0. After the base address is moved into eax, we can see the character ‘a’ (61h) moved into offset 0x14. The instructions fld and fstp are used to load and store the double at offset 0x18. At 0x00401017 we can see the initialization of the loop counter *i* to zero, followed by a jump to a comparison between *i* and 5 in order to execute the loop code. At (2), each iteration of the loop code will increase ecx by one (with ecx starting out as 0) and therefore each value of edx inserted into the integer array will occur at offsets 0x0, 0x4, 0x8, 0xC, and x010.

Define structures in IDA using [T]

Linked Lists

```c
struct node
{
   int x;
   struct node * next;
};

typedef struct node pnode;

void main()
{
   pnode * curr, * head;
   int i;
   head = NULL;

   for(i=1;i<=10;i++) ❶
   {
      curr = (pnode *)malloc(sizeof(pnode));
      curr->x = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr) ❷
   {
      printf("%d\n", curr->x);
      curr = curr->next ;
   }
}
```

```asm
0040106A        mov     [ebp+var_8], 0
00401071        mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078        cmp     [ebp+var_C], 0Ah
0040107C        jg      short loc_4010AB
0040107E        mov     [esp+18h+var_18], 8
00401085        call    malloc
0040108A        mov     [ebp+var_4], eax
0040108D        mov     edx, [ebp+var_4]
00401090        mov     eax, [ebp+var_C]
00401093        mov     [edx], eax ❶
00401095        mov     edx, [ebp+var_4]
00401098        mov     eax, [ebp+var_8]
0040109B        mov     [edx+4], eax ❷
0040109E        mov     eax, [ebp+var_4]
004010A1        mov     [ebp+var_8], eax
004010A4        lea     eax, [ebp+var_C]
004010A7        inc     dword ptr [eax]
004010A9        jmp     short loc_401078
004010AB loc_4010AB:
004010AB        mov     eax, [ebp+var_8]
004010AE        mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1        cmp     [ebp+var_4], 0 ❸
004010B5        jz      short locret_4010D7
004010B7        mov     eax, [ebp+var_4]
004010BA        mov     eax, [eax]
004010BC        mov     [esp+18h+var_14], eax
004010C0        mov     [esp+18h+var_18], offset aD ; "%d\n"
004010C7        call    printf
004010CC        mov     eax, [ebp+var_4]
004010CF        mov     eax, [eax+4]
004010D2        mov     [ebp+var_4], eax ❹
004010D5        jmp     short loc_4010B1 ❺
```

On the left we see a node structure defined which contains an integer and a pointer to the next node in the linked list. At (1) a for loop creates 10 nodes and links them, at (2) a while loop iterates over the list printing each node's value (in descending order from 10 to 1).

On the right we can see two variables var_8 representing *head* set to 0 (null), and var_C representing the for loop counter *i* set to 1. After the call to malloc, a pointer to the node structure is passed into var_4. At (1) the value of *i* (var_C) is passed into the first variable in the struct (var_4), which correlates to the part of the loop that sets *curr->x = i;* Following this at (2), we can see the value of *head* (var_8) passed into the second variable in the structure, which correlates to *curr->next = head;* in the code on the left. We then observe *head* (var_8) being set to *curr* (var_4) and the loop counter being incremented.

Once the for loop exits, *curr* (var_4) is set to *head* (var_8) and at (3) *curr* (var_4) is compared to 0 (the initial value of *head*). Each iteration of the while loop updates with moving the *next* variable of the node structure into *curr* at (4) and jumping back to the comparison at (5).

Realizing that (var_4) contains a pointer to another struct, which must also have a pointer to another struct, is essential to recognizing the linked nature of the list.

Resources

The C Programming Language book by Brian Kernighan and Dennis Ritchie