x86 Disassembly/Data Structures

Data Structures

Few programs can work by using simple memory storage; most need to utilize complex data objects, including **pointers**, **arrays**, **structures**, and other complicated types. This chapter will talk about how compilers implement complex data objects, and how the reverser can identify these objects.

Arrays

Arrays are simply a storage scheme for multiple data objects of the same type. Data objects are stored sequentially, often as an offset from a pointer to the beginning of the array. Consider the following C code:

```
x = array[25];
```

Which is identical to the following asm code:

```
mov ebx, $array
mov eax, [ebx + 25]
mov $x, eax
```

Now, consider the following example:

```
int MyFunction1()
{
   int array[20];
   ...
```

This (roughly) translates into the following asm pseudo-code:

```
:_MyFunction1
push ebp
mov ebp, esp
sub esp, 80 ;the whole array is created on the stack!!!
lea $array, [esp + 0] ;a pointer to the array is saved in the array variable
...
```

The entire array is created on the stack, and the pointer to the bottom of the array is stored in the variable "array". An optimizing compiler could ignore the last instruction, and simply refer to the array via a +0 offset from esp (in this example), but we will do things verbosely.

Likewise, consider the following example:

```
void MyFunction2()
{
   char buffer[4];
   ...
```

This will translate into the following asm pseudo-code:

```
:_MyFunction2
push ebp
mov ebp, esp
sub esp, 4
lea $buffer, [esp + 0]
...
```

Which looks harmless enough. But, what if a program inadvertantly accesses buffer[4]? what about buffer[5]? what about buffer[8]? This is the makings of a buffer overflow vulnerability, and (might) will be discussed in a later section. However, this section won't talk about security issues, and instead will focus only on data structures.

Spotting an Array on the Stack

To spot an array on the stack, look for large amounts of local storage allocated on the stack ("sub esp, 1000", for example), and look for large portions of that data being accessed by an offset from a different register from esp. For instance:

```
:_MyFunction3
push ebp
mov ebp, esp
sub esp, 256
lea ebx, [esp + 0x00]
mov [ebx + 0], 0x00
```

is a good sign of an array being created on the stack. Granted, an optimizing compiler might just want to offset from esp instead, so you will need to be careful.

Spotting an Array in Memory

Arrays in memory, such as global arrays, or arrays which have initial data (remember, initialized data is created in the .data section in memory) and will be accessed as offsets from a hardcoded address in memory:

```
:_MyFunction4
push ebp
mov ebp, esp
mov esi, 0x77651004
mov ebx, 0x00000000
mov [esi + ebx], 0x00
```

It needs to be kept in mind that structures and classes might be accessed in a similar manner, so the reverser needs to remember that all the data objects in an array are of the same type, that they are sequential, and they will often be handled in a loop of some sort. Also, (and this might be the most important part), each elements in an array may be accessed by a *variable offset from the base*.

Since most times an array is accessed through a computed index, not through a constant, the compiler will likely use the following to access an element of the array:

```
mov [ebx + eax], 0x00
```

If the array holds elements larger than 1 byte (for char), the index will need to be multiplied by the size of the element, yielding code similar to the following:

```
mov [ebx + eax * 4], 0x11223344   # access to an array of DWORDs, e.g.    arr[i] = 0x11223344
...
mul eax, $20                      # access to an array of structs, each 20 bytes long
lea edi, [ebx + eax]              # e.g.     ptr = &arr[i]
```

This pattern can be used to distinguish between accesses to arrays and accesses to structure data members.

Structures

All C programmers are going to be familiar with the following syntax:

```
struct MyStruct
{
   int FirstVar;
   double SecondVar;
   unsigned short int ThirdVar;
}
```

It's called a **structure** (Pascal programmers may know a similar concept as a "record").

Structures may be very big or very small, and they may contain all sorts of different data. Structures may look very similar to arrays in memory, but a few key points need to be remembered: structures do not need to contain data fields of all the same type, structure fields are often 4-byte aligned (not sequential), and each element in a structure has its own offset. It therefore makes no sense to reference a structure element by a variable offset from the base.

Take a look at the following structure definition:

```
struct MyStruct2
{
   long value1;
   short value2;
   long value3;
}
```

Assuming the pointer to the base of this structure is loaded into ebx, we can access these members in one of two schemes:

```
;data is 32-bit aligned
[ebx + 0] ;value1
[ebx + 4] ;value2
[ebx + 8] ;value3
```

```
;data is "packed"
[ebx + 0] ;value1
[ebx + 4] ;value2
[ebx + 6] ;value3
```

The first arrangement is the most common, but it clearly leaves open an entire memory word (2 bytes) at offset +6, which is not used at all. Compilers occasionally allow the programmer to manually specify the offset of each data member, but this isn't always the case. The second example also has the benefit that the reverser can easily identify that each data member in the structure is a different size.

Consider now the following function:

```
:_MyFunction
push ebp
mov ebp, esp
lea ecx, SS:[ebp + 8]
mov [ecx + 0], 0x0000000A
mov [ecx + 4], ecx
mov [ecx + 8], 0x0000000A
mov esp, ebp
pop ebp
```

The function clearly takes a pointer to a data structure as its first argument. Also, each data member is the same size (4 bytes), so how can we tell if this is an array or a structure? To answer that question, we need to remember one important distinction between structures and arrays: the elements in an array are all of the same type, the elements in a structure do not need to be the same type. Given that rule, it is clear that one of the elements in this structure is a pointer (it points to the base of the structure itself!) and the other two fields are loaded with the hex value 0x0A (10 in decimal), which is certainly not a valid pointer on any system I have ever used. We can then partially recreate the structure and the function code below:

```
struct MyStruct3
{
   long value1;
   void *value2;
   long value3;
}

void MyFunction2(struct MyStruct3 *ptr)
{
   ptr->value1 = 10;
   ptr->value2 = ptr;
   ptr->value3 = 10;
}
```

As a quick aside note, notice that this function doesn't load anything into eax, and therefore it doesn't return a value.

Advanced Structures

Lets say we have the following situation in a function:

```
:MyFunction1
push ebp
mov ebp, esp
mov esi, [ebp + 8]
lea ecx, SS:[esi + 8]
...
```

what is happening here? First, esi is loaded with the value of the function's first parameter (ebp + 8). Then, ecx is loaded with a pointer to the offset +8 from esi. It looks like we have 2 pointers accessing the same data structure!

The function in question could easily be one of the following 2 prototypes:

```
struct MyStruct1
{
  DWORD value1;
  DWORD value2;
  struct MySubStruct1
  {
     ...
```

```
struct MyStruct2
{
   DWORD value1;
   DWORD value2;
   DWORD array[LENGTH];
   ...
```

one pointer offset from another pointer in a structure often means a complex data structure. There are far too many combinations of structures and arrays, however, so this wikibook will not spend too much time on this subject.

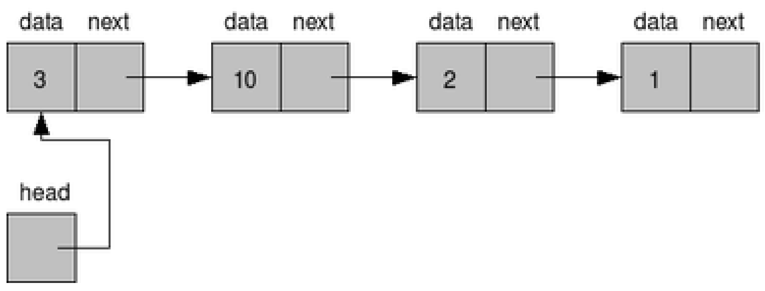
Identifying Structs and Arrays

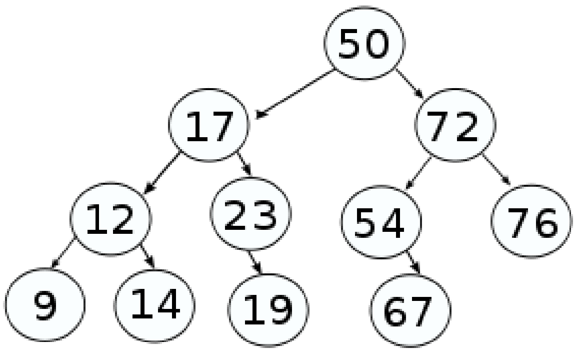
Array elements and structure fields are both accessed as offsets from the array/structure pointer. When disassembling, how do we tell these data structures apart? Here are some pointers:

1. Array elements are not meant to be accessed individually. Array elements are typically accessed using a variable offset
2. Arrays are frequently accessed in a loop. Because arrays typically hold a series of similar data items, the best way to access them all is usually a loop. Specifically, `for(x = 0; x < length_of_array; x++)` style loops are often used to access arrays, although there can be others.
3. All the elements in an array have the same data type.
4. Struct fields are typically accessed using constant offsets.
5. Struct fields are typically not accessed in order, and are also not accessed using loops.
6. Struct fields are not typically all the same data type, or the same data width

Linked Lists and Binary Trees

Two common structures used when programming are linked lists and binary trees. These two structures in turn can be made more complicated in a number of ways. Shown in the images below are examples of a linked list structure and a binary tree structure.





Each node in a linked list or a binary tree contains some amount of data, and a pointer (or pointers) to other nodes. Consider the following asm code example:

```
loop_top:
cmp [ebp + 0], 10
je loop_end
mov ebp, [ebp + 4]
jmp loop_top
loop_end:
```

At each loop iteration, a data value at [ebp + 0] is compared with the value 10. If the two are equal, the loop is ended. If the two are not equal, however, the pointer in ebp is updated with a pointer at an offset from ebp, and the loop is continued. This is a classic linked-loop search technique. This is analagous to the following C code:

```
struct node
{
  int data;
  struct node *next;
};

struct node *x;
...
while(x->data != 10)
{
  x = x->next;
}
```

Binary trees are the same, except two different pointers will be used (the right and left branch pointers).