

Data structure in assembly

Making sense of base + displacement

© Government of Canada

This document is the property of the Government of Canada. It shall not be altered, distributed beyond its intended audience, produced, reproduced or published, in whole or in any substantial part thereof, without the express permission of CSE.



Communications
Security Establishment

Centre de la sécurité
des télécommunications

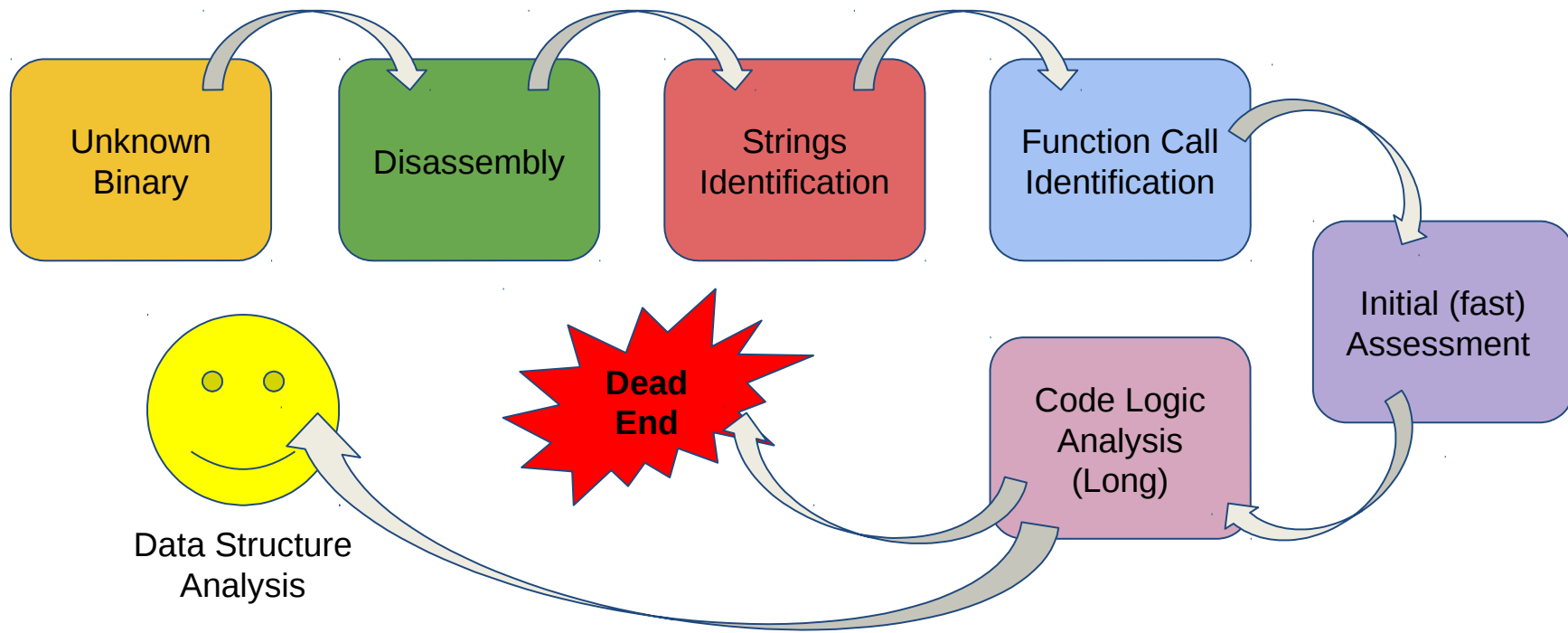
PAGE

Canada

Let's talk about the software reverse engineering process (SRE) for a moment...



Typical SRE project (static analysis - rough)



Code is typically organised around data structures!

You need to be able to find and make sense of these

© Government of Canada

This document is the property of the Government of Canada. It shall not be altered, distributed beyond its intended audience, produced, reproduced or published, in whole or in any substantial part thereof, without the express permission of CSE.



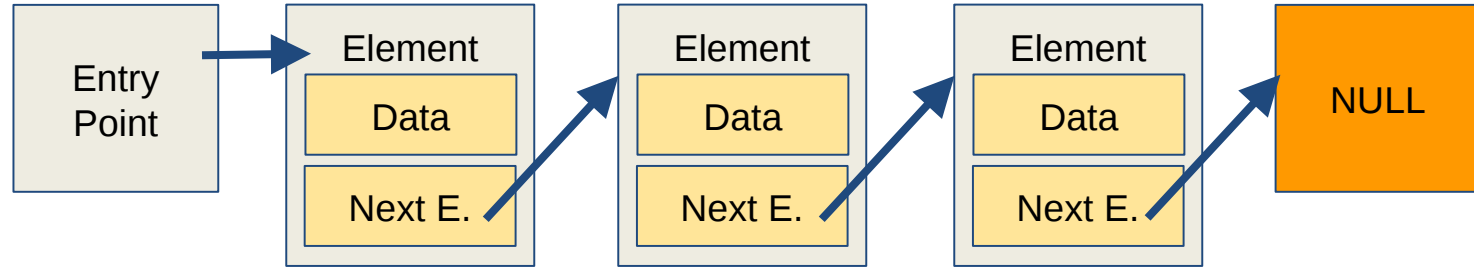
Communications
Security Establishment

Centre de la sécurité
des télécommunications

PAGE

Canada

Linked list - A quick review



Conceptually, a linked list must have an **entry point** and an **exit point**. Each element is an individual data structure independent of another. Contrary to a classical array, linked list **elements** are often **located** in **non contiguous memory**. Hence the **need** for “pointers” to **next element**. Operations on these need to be carefully planned as a **break in the “link”** between elements could and will result in **loss of data and memory leaks**.

Linked list - C

```
#include <stdio.h>
#include <stdlib.h>

struct element{
    int data;
    struct element *next;
};

typedef struct element IntLinkedList;

int main(int argc, char* argv[]){

    IntLinkedList *first = (IntLinkedList*)malloc(sizeof(IntLinkedList));
    first->data = 7;
    first->next = NULL;

    printf("First element data is %d\n", first->data);
    free(first);
    return 0;
}
```

In C, structures have definitions that can be used in accessing data element and next element pointers easily.

That definition is, however, only useful for the programmer and the compiler when generating assembly code.

Structures definitions are lost when code is translated to assembly by the compiler.



Linked list - Lost definition

```

main:
push    call    malloc
mov     lea     ecx, [.L.str]
sub     mov     dword ptr [ebp - 16], eax
mov     mov     eax, dword ptr [ebp - 16]
mov     mov     dword ptr [eax], 7
mov     mov     eax, dword ptr [ebp - 16]
mov     mov     dword ptr [eax + 4], 0
mov     call    mov     eax, dword ptr [ebp - 16]
mov     mov     eax, dword ptr [eax]
mov     mov     dword ptr [esp], ecx
mov     mov     dword ptr [esp + 4], eax
mov     call    printf
mov     mov     ecx, dword ptr [ebp - 16]
mov     mov     dword ptr [esp], ecx
mov     mov     dword ptr [ebp - 24], eax
call    free
xor     eax, eax
add     esp, 40
pop     ebp
ret
  
```

In this example, the **pointer to the buffer** allocated for the first element of the structure is **located at ebp-16**. Here is the definition of our original structure:

```

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

The original code was using “data” and “next” to refer to various parts of the element structure. This has been lost and a **base + displacement** memory access is now **required** to reach the various parts of the structure.

As you can see, any structure can become a real challenge in conducting SRE... Let's do a quick IDA pro demonstration...



Are there any questions?

Understanding data structures is essential for the next practical exercise and to understand the next topic we will cover.

© Government of Canada

This document is the property of the Government of Canada. It shall not be altered, distributed beyond its intended audience, produced, reproduced or published, in whole or in any substantial part thereof, without the express permission of CSE.

