**Lab Title: Understanding Memory Layout of a Process**

https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/#install-windbg-directly

**Objective:**

* Understand different segments of a process memory layout.
* Analyze stack, heap, data, and text areas using WINDBG.
* Explore activation records and calling conventions (\_\_cdecl, \_\_stdcall, \_\_thiscall).

**Part 1: Understanding Memory Segments**

**1.1 Process Memory Layout Overview**

Each running process consists of the following segments:

* **Text Segment**: Stores the executable code.
* **Data Segment**: Stores global and static variables.
* **Heap Segment**: Used for dynamic memory allocation.
* **Stack Segment**: Stores function calls and local variables.

**1.2 Hands-on: Observing Memory Layout**

**Step 1: Write a Sample C Program**

Create a file memory\_layout.c:

#include <stdio.h>

#include <stdlib.h>

int global\_var = 10; // Stored in the Data segment

static int static\_var = 20; // Also stored in Data segment

void function() {

int stack\_var = 30; // Stored in Stack segment

printf("Stack Variable: %d\n", stack\_var);

}

int main() {

int local\_var = 40; // Stored in Stack segment

int \*heap\_var = (int \*)malloc(sizeof(int)); // Stored in Heap segment

\*heap\_var = 50;

printf("Text Segment: %p\n", (void \*)main);

printf("Data Segment: %p, %p\n", (void \*)&global\_var, (void \*)&static\_var);

printf("Heap Segment: %p\n", (void \*)heap\_var);

printf("Stack Segment: %p\n", (void \*)&local\_var);

function();

free(heap\_var);

return 0;

}

**Step 2: Compile and Run the Program**

gcc memory\_layout.c -o memory\_layout

./memory\_layout

Observe the memory addresses printed.

**Part 2: Analyzing Memory Using WINDBG**

**2.1 Setting Up WINDBG**

1. Open WINDBG as administrator.
2. Attach WINDBG to the running process:

.attach <process\_id>

**2.2 Commands for Memory Inspection**

* View loaded modules: lm
* View memory layout: !address
* View stack information: k
* View heap allocations: !heap

Run these commands and analyze the output.

**Part 3: Activation Records and Calling Conventions**

**3.1 Understanding Activation Records**

Each function call creates an activation record containing:

* **PROLOG**: Setup of the stack frame.
* **B-LOGIC**: Function logic execution.
* **EPILOG**: Stack cleanup before returning.

**3.2 Hands-on: Observing Calling Conventions**

Create calling\_convention.c:

#include <stdio.h>

void \_\_cdecl cdecl\_function() { printf("CDECL Function\n"); }

void \_\_stdcall stdcall\_function() { printf("STDCALL Function\n"); }

void \_\_thiscall thiscall\_function() { printf("THISCALL Function\n"); }

int main() {

cdecl\_function();

stdcall\_function();

thiscall\_function();

return 0;

}

Compile and analyze the assembly using:

gcc -S calling\_convention.c -o calling\_convention.s

Observe differences in stack management.

**Conclusion:**

* Memory segments have distinct roles in process execution.
* WINDBG is useful for debugging memory layouts.
* Different calling conventions impact stack management.