In-depth Understanding of

Memory Leak
and
Dangling Pointer

When you are on a journey of creating a custom memory allocator for a custom memory management system, it is important to analyze and understand the key problems that would arise and address them in the first place.

Yashwanth naidu Tikkisetty

In this article, we will be diving into the most familiar, well heard yet most important problems, Memory Leak and dangling Pointer.

We won't just be discussing about the problems but also the possible scenarios it might arise and corresponding solutions for them.

P.S: When I talk about creating a memory allocator and memory management systems, its not about calling the functions that already exits; i.e

Its not **yourMalloc() →malloc()→ memory allocated.**

But its, **yourMalloc()→(inner implementation with optimizations for allocation) → memory allocated**.

When we talk about Dynamic Memory Allocation, in C/C++ we implement them using malloc() and free(). Malloc() to allocate the memory and free() to free up that allocated memory.

Memory Leak

What is Memory Leak and how does memory leak happen?

A memory leak occurs when a program allocates memory by reserving a portion of the computer's memory for its use, but then fails to release it back to the system. This means the memory remains 'occupied' and is not available for other processes or applications, even though it is no longer needed by the program.

Memory Leaks most certainly occur when we use Dynamic Memory Allocation. A memory leak occurs when the programmer fails to properly release this memory.

Consider the following example:

```c
#include <stdlib.h>

void functionWithLeak() {
    int *ptr = malloc(sizeof(int));  // Memory is allocated
    *ptr = 10;                       // The allocated memory is used
    // Memory is not freed - leak occurs
}

int main() {
    for (int i = 0; i < 100; i++) {
        functionWithLeak();
    }

    // Ideally, we should free the memory here, but it's already lost
    return 0;
}
```

It's a simple read through code. You can clearly see that The crucial issue here is, the allocated memory is never freed. The pointer ptr goes out of scope at the end of each functionWithLeak() call, and the reference to the allocated memory is lost.

Since there's no way to access this memory again, it cannot be freed later. This results in a memory leak.

How to prevent it?

Simply free the memory after usage. Something as simple as adding the following line for the above code would prevent the memory leak.

```c
void functionWithLeak() {
    int *ptr = malloc(sizeof(int));
    *ptr = 10;
    free(ptr);  // Frees the allocated memory, preventing the leak
}
```

The above example is a simple short one. Now imagine the same problem for a scenario, you are coding a solution that involves multi-dimensional matrix calculation and now the code size if of some 30,000 lines. It's got multiple files and multiple function calls.

Why are memory leaks problematic?

- As memory leaks accumulate, they consume more and more of the system's available memory. This can slow down not just the leaking application but potentially other applications running on the same system.

- In severe cases, memory leaks can lead to system instability or crashes. When the system runs out of memory, it may start swapping heavily or fail to allocate memory for critical operations, leading to failures.

- Memory leaks waste system resources. They tie up memory that could otherwise be used for other processes or applications, reducing the overall efficiency of the system.

Example case:

```c
#include <stdlib.h>

int main() {
    while (1) {
        int *ptr = malloc(1024 * sizeof(int));  // Allocate 4KB (assuming int is 4 bytes)
        if (ptr == NULL) {
            exit(1);  // Exit if memory allocation fails
        }
        // Simulate doing something with the memory
        *ptr = 10;
        // Memory is not freed - leak occurs
    }
    return 0;
}
```

Each iteration here allocates 4K bytes of memory but is never freed leading to a memory leak. As the program continues to run, it will consume more and more memory, leaving less available for other applications and system processes.

This can lead to system slowdowns, as the operating system may start using swap space (if available) to compensate for the lack of physical memory.

If the system runs out of memory (including swap space, if used), the **malloc** function will fail to allocate new memory and return **NULL**. In this example, the program will exit in such a case, but in a real-world scenario, inadequate handling of such failures can lead to application or system crashes.

Use tools like Valgrind, AddressSanitizer, BoundsChecker, Deleaker, Dr. Memory, memwatch to check for memory leaks.

If there is a memory leak on application level, the application may be crashed, but if the leak is on Kernel level, the OS itself will fail. In simple terms,

When **Memory leak** occurs in,

Short-lived applications, the application crashes but the OS recollects the lost memory.

Long-lived applications, consumes up all the RAM and would lead to crash and abnormal behavior.

Kernel level, Fatal for OS and System instability of system.

Consider a real world example, Elevator control software. In this scenario, a memory leak occurred due to a programming oversight where memory allocated to remember the floor number was not released if the elevator was already on the requested floor. While such a case might not have immediate effects, as the elevator would likely have enough spare memory to handle several instances, over time, the cumulative effect of these leaks could be substantial. Eventually, the elevator could run out of memory, potentially after months or years, leading to its

failure to respond to move requests. This could result in situations where people are unable to call the elevator or, if inside, become trapped due to the doors failing to open.

Now about Dangling pointer.

A dangling pointer arises in programming when a pointer that had been pointing to a memory location continues to point to that location after the memory has been deallocated or freed. This can lead to various types of programming errors and vulnerabilities, as the memory location may be allocated again for another use and may contain different data.

- When a pointer is pointing to a memory location, and the memory location is deallocated (using functions like **free** in C), the pointer becomes a dangling pointer.
- If the deallocated memory is reallocated for a different purpose, any attempt to access the dangling pointer will lead to undefined behavior because it points to the wrong data.

There are even types of Dangling pointers:

- **Wild Pointers**: These are uninitialized pointers that can point to any memory location, and sometimes become dangling unintentionally.
- **Pointers to Stack Memory**: When a function returns, all stack-allocated variables (including pointers) are freed. A pointer returned by a function that points to a stack variable becomes a dangling pointer.
- **Pointers to Heap Memory**: After dynamic allocation (using **malloc**, for instance), if the memory is freed and the pointer isn't set to **NULL**, it becomes a dangling pointer.

Dangling pointers can lead to security vulnerabilities like buffer overflows, which can be exploited by attackers to execute arbitrary code.

Since the pointer may access some other allocated space, it can lead to unexpected changes in the data, leading to program errors and data corruption.

Accessing a dangling pointer can cause program crashes or other forms of undefined behavior, making the program unstable.

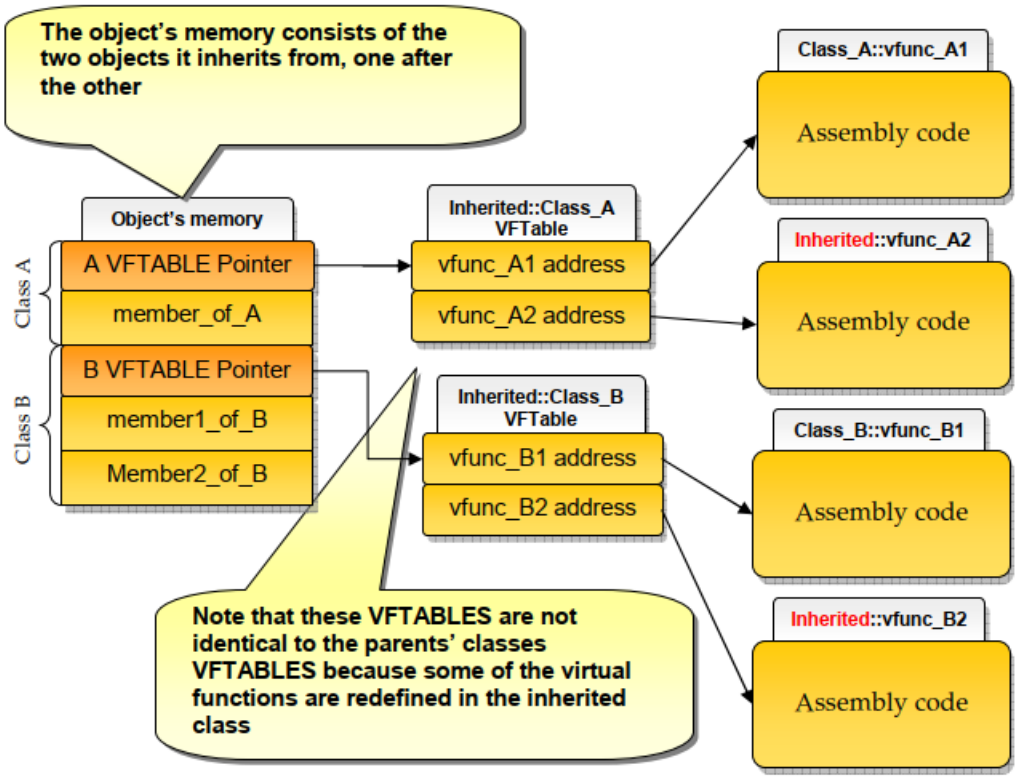
Just like a Buffer over flow exploit, if not taken care, there would be an exploit with Dangling pointers.

Watchfire IIS 5.1 dangling pointer vulnerability: "In December 2005, a Dangling Pointer was found in the IIS web server version 5.1, which is the default IIS server that ships Microsoft Windows XP SP2. A slight variation of this bug was discovered while scanning the IIS server with AppScan: Watchfire's web application security assessment tool. This scan resulted in a crash of the IIS server process(inetinfo.exe)."

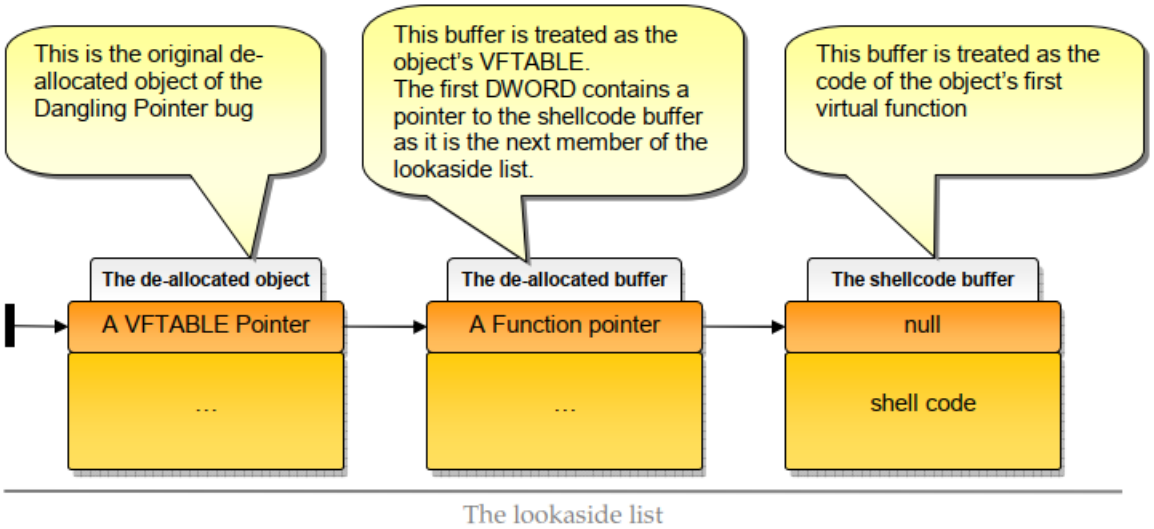
What was that vulnerability about?

The IIS server uses a pointer to an object that is de-allocated when a sequence of requests is sent to the server

The High level sequence



The Lookaside list:



"The first DWORD of the de-allocated object, which usually points to the object's VFTABLE. is cleared when the de-allocation is called. The next time the server tries to call a virtual function of the object, the application will try to find the VFTABLE at location 0x00000000 which will result in a crash.

Using the above technique, (from the given link), a group was able to force the server to reallocate memory in the same location and set its memory with their own data. This allowed them to inject pieces of code to be executed on the server using the "System" credentials. "

The above data and images: https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

Hope you had a detailed understanding about Dangling pointer and Memory Leak. In further articles, we will be exploring more about building our custom memory management system.

~~~~~~

An Article by Yashwanth Naidu Tikkisetty

(Yash)

~~~~~~