International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 05 | May 2022 www.irjet.net p-ISSN: 2395-0072

Memory Management in Trading Platforms

Pratik Joshi

Abstract – *With the increasing volatility in the market we have seen a sharp rise in number of trades occurring in the market. These platforms are designed to handle huge volume of trades thanks to the underlying technology. The memory management forms a base of these highly efficient algorithms. The technique in which OS loads programs into memory so that it can execute several such processes in parallel is a key functionality of the CPU. This paper will talk about common techniques used by trading platforms for managing memory of their applications. With high volume there is always a risk of crashing the application in the middle of a trading day. An immediate concern in most cases of a memory leak is unwanted behavior by the application that causes unexpected crashes, which could lead to damaged relationships with end users or clients who use the application. Worse yet, if an attacker were to figure out a way to trigger a memory leak, it could allow them to perform a denial-of-service attack. As such, memory leaks represent a combination of performance and security concerns. It is therefore recommended to keep memory management in mind while designing trading systems where computation is of essence. Memory leaks can be difficult to avoid, but there are plenty of tools out there that can help. It's recommended that you utilize them regardless of which programming language you prefer. A compromised memory could lead to denial-of-service or corrupt the data of your application. Some common techniques will be covered in this paper using which the application designer can make a better system.*

Key Words: Fragmentation, paging, buffer overflow, memory allocation, high frequency trading, in-memory computing.

1. INTRODUCTION

Financial industry heavily uses high-frequency trading in which the securities are traded on the financial markets using high-speed rules-based strategies, and numerous simultaneous trades – with all the decisions driven by computerized, quantitative models. The computer program analyzes the market data and trend and computes a buy or sell trade or perform other financial services. They compute the data points to predict the market movement and act. These trades are 10 times faster than time taken by a trader doing this manually. These techniques prove to be handy during market instability, market volatility or financial crisis. Because of the time constraint the systems act much quickly thereby optimizing the use of technology on trading platforms.

Memory management represents a vital part of secure application development. Proper memory management is like good personal hygiene. We are physically healthier when we practice proper hygiene. Similarly, applications perform better when memory use is properly allocated. This paper demonstrates the risks of poor memory hygiene, including buffer overflow, memory leaks, memory allocation, and nulling out pointers. By the end of this paper, you will have a better understanding of why these processes could create security risks and how to avoid them.

Memory is a collection of data like instructions for processor or large array of data. During execution of programs CPU uses memory to hold data, fetches instructions from memory. To optimize this flow and make the process efficient in multi programming environment we need to ensure efficient utilization of memory. In case of high frequency trading platform where there is huge volume every millisecond of reading data from memory matters.

1. Overview of Compilers

Compilers for C and C++ parse source code and emit instructions for the CPU they are targeting. These instructions are commonly called assembly instructions. Since these instructions are for the CPU itself, with no intermediate layer, they are described as low-level instructions. Other programming languages, like C# or Java, have an entire middle layer between the compiler-generated instructions and what is sent to the CPU, which helps prevent mistakes. These two languages are thought of as higher-level programming languages because they do more behind the scenes to add safety and security to an application. Conversely, the safety of an application built with C/C++ is left in the developer's hands with very few safeguards in place to prevent potential bad code from executing on the CPU.

Static and Dynamic Loading:

A loader is used to load a process into the main memory. A static loader would generally load the entire routine into a fixed address. Dynamic loader loads a routine only after it has been called.

Swapping:

When a process executes it must have resided in memory. Swapping is a process responsible for swapping routines into main memory from secondary memory. Swapping allows high number of processes to be run by efficiently using algorithm and fitting into memory.

Memory allocation:

Memory allocation is the process of setting aside sections called partitions of memory in a program to be used to store data for a process. It has two techniques, namely static and dynamic allocation.

Fragmentation:

When the process is loaded and removed after execution it leaves behind the memory blocks which cannot be allocated to the processes due to their small size and the blocks remain unused.

Paging:

Processes are divided into pages. One page of the process is to be stored in one of the frames in the memory. Paging technique is used to avoid using the contiguous allocation of physical memory.

2. Buffer Overflow

A buffer overflow is simply allocating an array of memory onto the call stack—the data structure where methods and functions are stored—and then overfilling it with more data than it was supposed to handle. The extra bytes written to memory spill over and overwrite adjacent memory, usually corrupting other stack-based variables.

Let's take an example for demonstrating buffer overflow with a username of 8 bytes and overflow of 2 bytes in severe cases, a buffer overflow will corrupt the call stack leading to a massive crash. Even worse, if an attacker has access to the source code, they could deduce a way to corrupt the call stack just enough to change the value of a variable that normal code could not reach, such as changing the privileges of a user to that of an admin.

Per the Open Web Application Security Project (OWASP), buffer overflow vulnerabilities typically occur in code that:

- Relies on external data to control its behavior
- Depends upon properties of the data that are enforced outside of the immediate scope of the code Is so complex that a programmer cannot accurately predict its behavior

Example:

The following code gathers input from the user and writes the input characters to a stack-based array. This stack-based array is a fixed length buffer and contains a password. When overfilled, it will corrupt another variable that determines if the user has admin privileges. Let's demonstrate how serious this can be:

```
int main ()
{
    int is_admin = 0xdeadbeaf;
    int buf [5];

    int* iter = buf;
    puts("Enter text. Press Enter to exit:");
    int c;
    do {
        c = getchar();
        *iter = c;
        iter++;
    } while (c != '\n');
    if (is_admin != 0xdeadbeaf)
        printf("Granted admin privileges");
    else
        printf("Access denied");
    return 0;
}
```

Run the application and enter a word with 8 or more letters.

For instance, enter in 8 characters of just the character '1', that is '11111111'.

When done typing, hit Enter to complete the input and finish. Note that the character for the Enter key is also returned in the string (which is '/n').

When the application starts, the memory layout of the stack variables may look like this (your compiler may be different):

```
0x010FF94C cccccccc
0x010FF950 cccccccc
0x010FF954 010ff960 <- iter
0x010FF958 cccccccc
0x010FF95C cccccccc
0x010FF960 cccccccc <- buf[0]
0x010FF964 cccccccc <- buf[1]
0x010FF968 cccccccc <- buf[2]
0x010FF96C cccccccc <- buf[3]
0x010FF970 cccccccc <- buf[4]
0x010FF974 cccccccc
0x010FF978 cccccccc
0x010FF97C deadbeaf <- is_admin
```

The left column is the memory address, and the right column is the data in the memory address in hexadecimal notation. In this situation, you can see how the memory address increments as it goes down. When the compiler creates or lays out the call stack, it puts the first variables declared at the bottom with higher memory addresses, and the last variables declared at the top with lower memory addresses. Incrementing a pointer moves it from top to bottom (as our code above does).

When a buffer overflows, data is written from areas starting in lower memory addresses, spilling over to areas of higher memory addresses. Notice how in the table there is a two-byte gap (in a 32-bit application) between the declaration of

the variables. This will vary depending on your compiler. When the do/while loop writes its first character: *iter = c, the memory looks like this:

```
0x010FF94C cccccccc
0x010FF950 cccccccc
0x010FF954 010ff960 <- iter
0x010FF958 cccccccc
0x010FF95C cccccccc
0x010FF960 00000031 <- buf[0]
0x010FF964 cccccccc <- buf[1]
0x010FF968 cccccccc <- buf[2]
0x010FF96C cccccccc <- buf[3]
0x010FF970 cccccccc <- buf[4]
0x010FF974 cccccccc
0x010FF978 cccccccc
0x010FF97C deadbeaf <- is_admin
```

When the do/while has iterated 5 times, it looks like this:

```
0x010FF94C cccccccc
0x010FF950 cccccccc
0x010FF954 010ff960 <- iter
0x010FF958 cccccccc
0x010FF95C cccccccc
0x010FF960 00000031 <- buf[0]
0x010FF964 00000031 <- buf[1]
0x010FF968 00000031 <- buf[2]
0x010FF96C 00000031 <- buf[3]
0x010FF970 00000031 <- buf[4]
0x010FF974 cccccccc
0x010FF978 cccccccc
0x010FF97C deadbeaf <- is_admin
```

As you can see, if it continues unchecked, it will overwrite the variables higher up in memory:

```
0x010FF94C cccccccc
0x010FF950 cccccccc
0x010FF954 010ff960 <- iter
0x010FF958 cccccccc
0x010FF95C cccccccc
0x010FF960 00000031 <- buf[0]
0x010FF964 00000031 <- buf[1]
0x010FF968 00000031 <- buf[2]
0x010FF96C 00000031 <- buf[3]
0x010FF970 00000031 <- buf[4]
0x010FF974 00000031
0x010FF978 00000031
0x010FF97C 00000031 <- is_admin
```

By the eighth iteration of the loop, the value of is_admin has been changed, and program flow will be altered. The program output looks like this:

```
Enter text. Press Enter to exit:
11111111
Granted admin privileges
```

This example shows how to corrupt a variable on the stack, but this same type of problem can occur for memory allocated on the heap as well. Though it may be harder to get variables so close to each other when allocated on the heap, the app may crash much later than a stack-based overflow, making it much harder to debug.

The simplest way to avoid buffer overflow issues is to use a modern programming language. Avoid C unless you have experience doing so. Even then, you should strongly consider switching to C++, which heavily minimizes dependence on C-based, stack-based buffers. In general, the more modern the language, the safer it is. Meaning, it might not expose such low-level memory management to the programmer.

Memory Leaks

A memory leak occurs when a developer fails to free an allocated block of memory when no longer needed. An application littered with memory leaks will eventually request a memory chunk and fail, because the address space is fragmented into tiny pieces.

A memory allocation in C++ looks like this:

```
int* foo = new int[456];
```

The leak happens when nothing more is done after the memory allocation. Pretty hard to spot, isn't it? Especially when the programmer forgets about it.

The simple solution looks like this:

```
delete [] foo;
```

When memory is allocated, it looks for a contiguous block of memory of a certain size. Any leaked memory that is not freed is unavailable for other memory and is blocked from being reallocated again. One memory leak may not be consequential, but if this happens enough, the application could crash.

Memory Leak Example

Here is a basic memory leak in C

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

#define LOOPS 10
#define MAXSIZE 256

int main(int argc, char **argv)

{
    int count = 0;
    char *pointer = NULL;

    for(count=0; count<LOOPS; count++) {
        pointer = (char *)malloc(sizeof(char) * MAXSIZE);
    }

    free(pointer);
    return count;
}
```

In this example, there are 10 allocations of size MAXSIZE. Every allocation, except for the last, is lost. If no pointer is pointed to the allocated block, it is unrecoverable during program execution. A simple fix to this trivial example is to place the free() call inside of the "for" loop.

3. CONCLUSIONS

In conclusion therefore, it is evident that memory management is one of the critical responsibilities of the operating system. Typically, primary memory is volatile as it holds the data and programs needed for processes to execute in the CPU while secondary memory provides long term data and program storage. The operating system assigns the responsibility of managing memory to the memory management unit (MMU). The OS ensures that programs and data are assigned and moved out of memory during program execution through the MMU which resides in the OS's kernel. When programs want to run in the CPU, processes must be swapped in and out of main memory. The swapping process creates holes that have the possibility of impairing the system's throughput. That is because swapping may cause internal or external fragmentation of the main memory. To improve throughput and minimize the effects of fragmentation, several memory placement techniques are used. These include the first fit policy and best fit policy. In first fit policy, the OS allocates a process to a hole that is first available so long as it can accommodate the process. Therefore, the allocation mechanism uses a process's index to allocate the process a position in the queue. However, the best fit policy can easily lead to the creation of many holes, impairing the efficiency of the OS.

The policy is since main memory is first scanned of all the holes that have been created and the hole that can fit the process's memory requirements is assigned. One of the algorithms that is used to assign processes memory holes is the round robin algorithm. These two allocation methods have been identified to be very efficient. However, for the OS to efficiently work it should allocate memory chunks to running programs. On the other hand, memory management cannot be complete if virtual memory is not considered. Virtual memory supports multiprogramming by allowing several resident programs to run at the same time. Allocating memory on the stack is easy to clean up afterwards, since the compiler does it for you. As the stack unwinds, the memory is automatically freed. Memory allocated on the heap is different; it is not automatically freed, and you must do it manually.

REFERENCES

[1] Breecher, J. Operating systems memory management. 2011. Web.

[2] A GridGain Systems In-Memory Computing White Paper

[3] Loepere, K. Mach 3 Kernel Principles. Open Software Foundation and Carnegie Mellon University, 1992. Web.RFC4120: The Kerberos Network Authentication Service (V5) [Applied Cryptography] Second Edition, Bruce Schneider

[4] Tanenbaum, Andrew, s. and Albert s. Woodhull. Operating systems design and Implementation. 2006, Prentice Hall. Web.

BIOGRAPHIES

I have been in Finance and Technology for over 9 years. At MarketAxess, I design solutions for our leading electronic trading platform for fixed-income securities. My team and I manage the market data and post-trade services for the global fixed-income markets. We are responsible to report trades to clearing houses in timely manner. I work on making the trading platform that sees on an average $300 billion monthly volume efficient and optimized.