Efficient and secure memory allocator

Roman Astrakhantsev  
Department of Cyber Security  
MIEM, HSE University  
Moscow, Russia  
rgastrakhantsev@edu.hse.ru

Mitin Arseny  
Department of Cyber Security  
MIEM, HSE UniversityMoscow, Russia  
avmitin@edu.hse.ru

*Abstract* — Memory allocation task have taken the first place by CPU usage in a number of applications. Different self-written memory managers tend to maximize the performance but still can lack security and be vulnerable. This paper focuses on proposing the efficient and secure allocator algorithm. Address randomization and “Split and Consume” strategy are the two major methods that help achieving such requirements. The results show that proposed memory manager can protect vulnerable programs form “use-after-free” attacks. Besides, it requires only 32 bits of additional metadata each allocation.

Keywords — /\* TO DO \*/

# Introduction

Modern applications cannot be imagined without processing a large amount of data. Memory allocation is one of the major challenges that developers face in programming. Most operation systems are not real time based, so any direct memory request to such an OS can interrupt evaluations for an undetermined period of time. Moreover, some applications that use custom memory managers in pursuit of efficiency can suffer from attacks like “use-after-free”. This study endeavors to propose the efficient, and secure algorithm of memory allocation. The address randomization and “Split and consume” strategy are the methods we used to achieve these results. // TO DO : improve introduction

The rest of the paper is organized as follows. In Section 2, we demonstrate some existing ideas and techniques in this area. Section 3 presents the methods for memory management and formulates the allocation algorithm.

# Related work

Since memory allocation problem is ubiquitous in computer science, this academic work to determine the efficient and secure allocation mechanism becomes more important these days. In the following we want to present an overview of related work in this area.

As aforementioned, the systems with insufficient user data validation can deal with dynamic memory storage. Dewey et al. [1] formulate the “use-after-free” vulnerability type and the conditions for its execution. The paper focuses on C++ com-piled binaries where the memory manager cannot find and update pointers to program objects when they are moved. This same vulnerability might be present in other languages. The authors bring the examples of code with attack proof of concept.

Further, Zeng et al. [2] classify different attack types into such systems. They noticed how programs might be patched to gain required defense against “use-after-free” type. However, the method they mentioned requires 64 bits of metadata for every memory allocation call and 4Kb of guard pages. In this work we propose the algorithm with less memory usage.

The other approach to prevent vulnerability was demonstrated and analyzed in [3]. It suggests randomizing the address space and returning random address pointers. Moreover, the authors mentioned that most operating systems use the current approach. With respect to our work, we also use randomization approach to obtain security.

According to [4], methods like TSLF are the most effective ones from the time complexity perspective. At the same time the additional memory usage is required due to the hash map. Moreover, Masmano et al. [5] introduce the TSLF algorithm that has O(1) time complexity and demonstrate the proof of such behavior. The authors also overviewed the segregated free list like method and compared it with the provided one. It is mentioned that such allocators do not use the hash map and therefore they are free of its memory. Our study introduces the time and memory balanced protocol of allocation based on segregated free list type.

Another question which concerns memory allocation is fragmentation. Zlatanov [6] overviewed this phenomenon and suggested determining a series of partition pools with block sizes in a geometric progression. We suppose that such approach tends to use memory inefficiently. Nevertheless, as mentioned in [7], the current phenomenon can be avoided. In addition, realizations of the best policies are already known and might be implemented on the client side.

In terms of safety and efficiency at the same time, Liu et al. [8] introduced so-called “SlimGuard” allocator that is designed to be secure and effective. The authors compared memory and time usage of SlimGuard with different state-of-the art memory management algorithms. Similar to this, we introduce lightweight allocator which performance still needs to be tested and compared with others.

As can be seen in the literature review above, state-of-the art memory management allocators lack either time/memory efficiency or attack protection. Only experimental methods try to approach the optimal state in both directions. In our study we have collected different ideas and proposed time and memory balanced allocator.

# Proposed Algorithm

## OS Memory management

Most of the operation systems already possess memory simple allocation mechanisms. Despite that, the consumed time can reach high values. Therefore, one of the ideas of memory management is to request a large chunk of memory from an OS and mark the memory inside the application program.

From the security perspective, most of the vulnerabilities are based on the fact that hackers can somehow predict the address of the working object. That is why the universal protection would be the address randomization within the preallocated chunk.

## Basic allocation

As was mentioned earlier, the basic principles are marking the large chunk of the data. The memory provided by OS can be divided into several parts by creating control blocks inside the allocated memory. The control block is a simple structure that holds the number of allocated bytes and the index of the previous control block. In term of C language, it can be represented as

struct control\_block  
{  
 void\* *previous*;  
 unsigned long long *size*;  
};

The *size* bytes of data should be stored right after the control block. On any memory allocation there are only 32 bits of additional metadata.

It can be seen that memory control blocks are united into the bidirectional list data structure. That is why on any memory request it is possible to look through all the allocated control blocks and quickly find the fitting one.

While processing the request and, the allocator should decide whether a particular control block is busy by the application or not. To understand which block is currently in use, it is possible to encode the lower bit of the *size* field as a hint. If it is not set, the block is free. And conversely if the lowest bit is set, the block is in use.

## Split and Consume

Efficient memory management implies that allocation process will result with enough memory units but not twice as more than requested. Therefore, it is natural to split big control blocks and return to the application the fitting one. The crucial part of the allocation security is to randomize the offset before the returning block (and insert another control block there if possible).

It is worth noting that various small allocations and deallocations can lower the memory capacity almost by a half. That is why merging two adjacent control blocks into one bigger are likely to improve the memory management. So, in this scheme, all allocations and deallocations would reset the state of the control block list.

The above manipulations of control blocks is called “Split and consume” strategy.

# Conclusion

Different applications that implement custom allocation mechanism might lack enough secure or time/memory efficiency. In this paper we have presented common allocation approaches and implemented new memory management algorithm that is balanced in both requirements. This can protect programs vulnerable to attacks like “use-after-free”, using only 32 bits of additional metadata on each allocation.

In the future we could develop a virtual laboratory for testing memory allocation mechanisms. This could help to measure the security and efficiency level of the current, existing, and future algorithms and compare them. Last but not least, proposed algorithm could be reconstructed from multithreaded perspective in order to support a wider spectrum of applications.

##### References

1. David Dewey, Bradley Reaves, P. Traynor “Uncovering Use-After-Free Conditions in Compiled Code” // International Conference on Availability, Reliabil-ity and Security, ARES, 2015
2. Qiang Zeng, Golam Kayas, Emil Mohammed, Lannan Luo, Xiaojiang Du, Junghwan Rhee “HeapTherapy+: Efficient Handling of (Almost) All Heap Vulner-abilities Using Targeted Calling-Context Encoding” // 49th Annual IEEE/IFIP In-ternational Conference on Dependable Systems and Networks (DSN), 2019
3. Jonathan Ganz, Sean Peisert “ASLR: How Robust Is the Randomness?” // IEEE Cybersecurity Development (SecDev), 2017
4. Paul R. WilsonMark S. JohnstoneMichael NeelyDavid Boles “Dynamic Storage Allocation A Survey and Critical Review” // International Workshop on Memory Management, 1995
5. M. Masmano U, I. Ripoll, A. Crespo, J. Real “TLSF: A new dynamic memory allocator for real-time systems” // Proceedings. 16th Euromicro Confer-ence on Real-Time Systems, 2004
6. Nikola Zlatanov “Dynamic Memory Allocation and Fragmentation” // ESC, 2015
7. Mark S. Johnstone, Paul R. Wilson “The Memory Fragmentation Problem: Solved?” // ACM SIGPLAN Notices, 1998
8. Beichen Liu, Pierre Olivier, Binoy Ravindran “SlimGuard: A Secure and Memory-Efficient Heap Allocator” // Middleware '19: Proceedings of the 20th In-ternational Middleware Conference, 2019