КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 4 (32), 2026
ISSN 2663 – 4023
DOI 10.28925/2663-4023.2026.32.1096
УДК 004.056

Myroslav Shpak
Bachelor's Degree Student, None, Full-time Student
Place: Kyiv National University of Construction and Architecture, Kiev, Ukraine
ORCID: 0009-0005-2372-8373
amurtasmail@gmail.com

Oleg Kurchenko
Candidate of Technical Sciences, Associate Professor, Department of Software Systems and Technologies, Faculty of Information Technologies, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
ORCID:0000-0002-3507-2392
kurol@ukr.net

Yurii Shcheblanin
Candidate of Technical Sciences, Senior Research Officer, Department of Cyber Security and Information Protection, Faculty of Information Technologies, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
ORCID: 0000-0002-3231-6750
shcheblanin.yurii@knu.ua

REDUCING THE VULNERABILITY LEVEL OF GARBAGE COLLECTION USING ARENA ALLOCATOR

Abstract. Memory allocation is substantially overcomplicated with commonly used self-managing background systems like: garbage-collection, smart pointers and RAII.
Oftentimes these systems and memory allocation paradigms cause performance and safety issues. In most scenarios (projects) a decent alternative to said allocation paradigms can be used to avoid the performance hit that garbage-collection and smart pointers can impose on the made software, and greater control can be achieved that can help avoid possible safety risks in between prune-and-sweep clean intervals of garbage collection.
Arena allocator is a pretty appealing allocation strategy that has relatively few mentions despite all the said benefits. With very concise implementation, Arenas can be very handy and make memory management trivial while keeping full control in hands of the developer. Arenas can be widely used in almost any project that revolves around real-time systems, user oriented software or inside highly optimized systems such as compilers.

Keywords: memory allocator; allocation strategy; memory garbage-collection; GC; arena allocator; use-after-free vulnerability.

INTRODUCTION

Overview. In the current landscape of programming languages, both interpreted and compiled, a dominant strategy of memory management is garbage-collection - an automatic background system responsible for performing system calls for memory allocation, keeping track of allocated objects (or chunks), and getting rid of them when memory is no longer used by the software. This system, unlike manual methods of memory allocation, doesn't require nearly any actions from the programmer side, which makes it a very appealing choice for prototyping or even building software with garbage collection built in as core dependency on which the software relies to function. The convenience of this system is undeniable, but despite the benefit of not taking responsibility for performing allocations, it has a set of flaws that are often ignored or significantly undervalued. Use of garbage collection can lead to

significant performance issues [1] with a wide variety of reasons, ranging from long times taken to compute references used, to being cache inefficient and having sparsely stored data. Also extensive use of garbage collection can lead to safety concerns or even high-level vulnerabilities, that can lead to arbitrary code execution.

Examples and concerns. Recent occasion with Redis has been caused by miss-handling an allocated object in custom implementation of Lua scripting programming language, which lead to having undiscovered (for a short period of time) of CVSS 10.0 [2], which can allow a hacker to escape sandboxed environment and do pretty much anything that user-space of operating system can allow, which is a long list of actions: from reading files, to sending data over network or even unfolding malicious code to take control of the machine, all of that due to a poor timing in-between garbage allocation cleaning intervals, a property of GC - that almost never can be found in any other memory management system.

The goal of this paper is to explain why Garbage collection isn't a best choice for general purpose software like desktop applications, servers and real-time systems (such as games), why it is slow, and how it can lead to vulnerabilities such as one described with custom version Lua programming language. Show how it can destroy performance in games or inside server infrastructure. And also provide a simpler alternative for memory management, that despite its manual nature – is very easy and safe to use, with given example on software that uses that strategy at its core. It's important to have all the necessary information in mind when choosing a technology for financial or public use, so that anyone who uses a product or a general use tool can have a safe and quick experience with it, a lot of effort is concentrated into different aspects of software engineering other than memory management, ever since the popularization of Java very few languages change the approach of using the Garbage Collector for their overall memory management, which seemingly accepted to be the best, however that does not hold true for many projects that became popular or used enough for a problem of memory management being apparent for a regular user in different forms, such as performance issues or safety concerns. In this paper a better alternative is proposed to provide a wider angle of view on memory management strategy and lead to better practices in software engineering and software safety.

THEORETICAL INTRODUCTION INTO THE PROBLEM

Part 1 – Introduction into memory management. Before understanding why garbage collection isn't the best, and why arenas are great in places where garbage collection isn't, it is necessary to first take a look at the concept of memory management.

Memory management, fundamentally at software level – a process of calling a system procedure (or performing a manual system call) to request a block of memory of asked size from the Operating System kernel, which in return is given as a pointer(memory address) to the beginning of requested block. Since a given block requested from the OS is reserved for the current process, up until its termination, naturally an additional step is required to tell the OS to unmap a memory under the given pointer, so that other processes can use it for their needs. If a programmer forgets to call an unmapping procedure on the allocated block of memory, and then further on loses a pointer to the allocated memory – a leak occurs.

This may sound as something of small importance for some simpler software out there, since the operating system automatically releases all the memory allocated by the program after it terminates, but in-fact it's a pretty huge issue that can lead to program crashing if uncontrolled memory leaks continue to happen. This especially holds true for software that runs continuously in a loop, since in that case a program can potentially endlessly occupy memory. As a software engineer, or just an executive programmer, the process of memory

management is just a sequence of decisions made to allocate needed memory, keep track of all the memory life-time either manual, or via a made system, and dispose of it after the memory is no longer needed by the program.

This implies the default or simple "style" of managing memory, the – "allocate, use, free" model. But this method is really error prone, since the programmer is expected to remember and acknowledge every single allocation that was made. With data-structures like trees, lists and partitions, or any other way of storing data in numerous blocks, an engineer or executive programmer has to keep track of hundreds, thousands or potentially millions of individual memory blocks. It's only a matter of time before a mistake is made and memory is leaked.

Thus, on top of the simple model of: allocate, use, free(unmap), numerous algorithms are built to perform memory allocations in semi-automatic or fully-automatic ways. For instance the C standard memory allocation function "malloc" is in-fact at least a ten thousand lines of code (defined by C standard) system designed to be flexible in most use-cases where a programmer would want to have a memory allocation to happen. All this leads to the topic of this research – Garbage collector.

Part 2 – Garbage Collector. Garbage collector is background system that runs along side user-written program, its job is to keep track of every allocation that is performed using it, and count (or mark) all the visible or accessible pointers to allocated blocks, and on set interval or some trigger event – call a sweep cycle which releases a memory from unused allocations.

The fundamental work cycle of garbage collector [3] consists of:

– User calls an allocation function, that takes a block out of the pool-list of non used nodes managed by the GC, marks it as used and the pointer to the allocated block is handed to the user.

– A memory is used, the allocator is aware of places where a memory block is used, and re-evaluates whether the memory is still accessible via reference(s) or a pointer.

– To perform a check either an algorithm is run on set intervals, or a special watch system attached on key events, like creating a variable or appending into it (when its array or some sort of object).

– When no more references remain, or the object is no longer accessible (a function has returned, and memory block is not mentioned anywhere) - a block is marked to be "sweeped", or in another words, memory should be freed and block marked as unused, so that GC can reuse it in next allocation, the visual of how that looks is shown on Figure 1.

When "sweep" happens all the unreachable or unmarked nodes are freed.

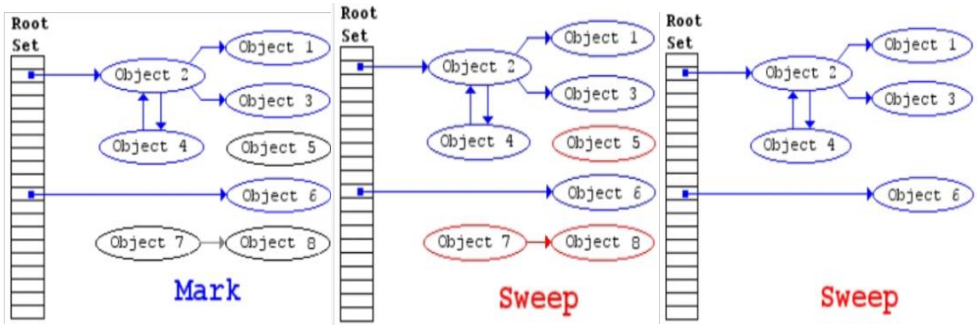

Figure 1 - Mark-Sweep cycles of Tracing garbage collector (in order)[4]

(a) Mark Stage from visibility; (b) Sweep stage from leftover nodes; (c) Sweep to free

Fundamentally garbage collector is not different from a regular algorithm that has trees, pool and queue data structures involved, what sets apart it is that often GC is implemented as core functionality of a language syntax and runtime system, as a result it is very easy to allocate memory from the heap using build-in mechanism of the language. There are a number of languages that have GC as one of their main features, just to name few: Java, JavaScript, C-Sharp, Haskell, D-Lang, V-Lang, Nim, Basic, Gleam, C++ (only from `C++11` to `C++23`), Python, Lua, and more.

In-fact , you can use garbage collector in languages that seemingly don't have it, such as: C, `C++`, Rust, Basic, or even assembly. Since garbage collector is just an algorithm that keeps track of pointers given by a memory allocation system call, nothing forbids wrapping regular allocation and release procedures into a GC system, and using it instead of manual allocations. Only difference from built-in GC's - is a requirement to manually call "sweep cycles" to find and release all the unused chunks of allocated memory.

Due to marketing and convenience (partially because of the immense success of Java) GC has become the most used allocation system , but often a lot of its drawbacks are either ignored or not even mentioned in the first place. Often times an excuse is used:

"A proper implementation of garbage collector has close to no issues in performing well majority of the time, and in the cases where manual intervention is needed, GC can allow a programmer to manually manage specific allocations to avoid performance issues. For safety reasons GC is the best because it completely eliminates human factors from being involved and solves a range of memory related bugs."

That argument does hold truth to some extent, however this point has few nuances:

– GC can be fast, but no matter the approach used, at some point it wouldn't be able to keep up with shear volume of memory being requested to keep track of (for instance in server applications or games).

– GC can be manually overridden by a programmer, however most of the time it is a pretty disruptive operation that can lead to unexpected behaviour or memory leaks (the human factor is always present even in scenarios that are taken with caution).

– In the cases when GC requires intervention quite frequently, it is pretty unclear why GC has to remain as the main method of performing memory management, when other approaches can be used.

– Garbage collector with sophisticated implementation can become a substantial dependency that can take a sizable amount of program background code or memory footprint, which may be a problem for platforms with smaller storage size or RAM.

– GC can have a variety of unpredictable behaviour in multi-threaded or concurrent environments.

– GC is still prone to having potential security holes that can be pretty tricky to account for ahead of time, such as use-after-free or undefined behaviour.

Also its important to understand that GC works as such, if looked at in vacuum, with no external environment in mind, however in cases when you have to use GC in pair with some external allocations happening, there is often no proper way to integrate that logic into GC, which can lead to hilarious situations - such as having a sizeable memory leaks in a "memory safe" language like Java. In some cases even native Java code(the one that doesn't use interface compiled native code like C or `C++`) can have leaks, which to be fair, is more of a language design flaw rather than fault of the GC.

To demonstrate the possible vulnerabilities, I would like to reference recent CVSS (Common vulnerability scoring system) level 10.0, that happens inside the custom implementation of Lua interpreter used by Redis - a database RAM caching software that acts

as middle phase between requiring to send a query to the database, and having it served from existing cache.

The essence of vulnerability is inside modified Lua's implementation of TString - a lua string structure that holds a pointer to the contents, length of the string, and some other miscellaneous data like hash and object properties. On the level of parsing the source text of lua script, a TString object is created, but is never added into the internal root GC node table - top level list of nodes that are currently used by the user in the program. As a result, whenever a sweep happens, the TString is freed, while being handed to the user via reference access. This causes a use-after-free, pretty standard error that is more characteristic to manual memory management scheme, however in this case the address space of object belongs to specifically garbage collector, and not arbitrary address on the heap, which means that with a little setup a malicious actor can get access to any kind of Lua object that can exist in the runtime of the script, via a faulty pointer, which allows to even call methods and functions, or access objects that usually are not permitted to be called by a script. This technically allows the user to escape the context of the running script and perform arbitrary code execution, perhaps even remotely. This also allows to acquire valuable data that can be allocated within the program, but not mentioned within the script, since GC can work for both actors - the script and the internal logic of the program.

It's important to note that getting the setup to acquire valuable data still is pretty difficult, however considering the nature of the Radis software being a big collection of SQL objects, there is enough attempts that can be performed with trail and error to hit something important – which is the reason why this exploit is rated 10.0 – highest possible score for the vulnerability.

To make a point about GC having rapidly declining performance in software, very little digging is required. Most remarkable (in my opinion) are these 3 examples, a Strategy that can solve all of the above problems is Arena allocator.

Example 1 – The RAM throughput clogging in the most popular game in the world (sale wise) - Minecraft. Minecraft is one of the (if not the most) popular sandbox games of all time, it has sold over 350 million copies (as of 2019 record logs), and that game is written in Java. Unfortunately very little effort is spent on making the game run well on any hardware, which pretty well demonstrates how GC can be a big detriment to game's performance. In short the game can allocate and/or free up to 1 GB/s of memory, which is a very high load on memory without an actual need to shuffle the allocations so fast. This can cause substantial lag that could be easily avoided with memory reuse.

Example 2 – A server lag spikes when serving user requests by one of the most popular social platforms for communication - Discord, dishing out Go programming language in favour of rust, due to having no perspective or room to improve further with GC as main bottleneck. In February 2020, Discord Inc. announced that they have migrated their server infrastructure from Go programming language to Rust. The reason was pretty simple, there were way too many incoming requests from clients to the server that they simply could not handle them without losses. The losses were caused by GC that was taking too long to process the market for sweep memory, which led to users experiencing noticeable packet loss and message delay. Dishing away Go has made the server much more stable, since Rust doesn't use GC to manage its allocations.

Example 3 – A custom voxel engine choosing to use Zig programming language due to lag spikes caused by GC sweeps. Author of open-source voxel sandbox game "CubyZ" have hit a limit of how much you can load a memory before causing GC to introduce lag spikes, in 2022 he published a devlog [5] talking about what would be his choice of language for future

iterations of the game, he talked about how `C++20` wouldn't be the best choice, also spoke about Rust, but generally settled on using Zig. In his future devlog's it is apparent that he has solved the issue, especially in his 2025 announcement video [6], where he stated that his project is open for the public to try out. In the demonstration he says that the game can run on immense render distances with stable frame rate, which definitely wasn't the case before the 2022's devlog explaining migration from Java to Zig.

Overall, it is pretty unreasonable to buy into GC as a magic bullet that solves the memory management in its entirety, even tho it can be very handy in some fields of software development, GC – is just a conceptually automatic memory management system, and sometimes it's very important to have finer control over memory allocations. Thus a good alternative to a fully automatic memory management system would be a semi-automatic one. A memory management strategy that reduces the most common mistakes made by a human programmer close to a zero, as well as allowing manual control of groups of data and their life-time in the program.

Part 4 – Arena Allocator. Arena allocator is a simple data structure that can be built on top of any general use allocator or implemented from scratch using only system specific procedures for allocating memory. The idea is that Arena allocator on memory allocation call, looks up in the chain of blocks, if requested size can be allocated in the current block, allocator increment a start pointer in that block, and gives that to the user, if not - that means remained memory in the block isn't big enough for the asked data, so we request another block from OS and add it to the chain (linked list) of the nodes, update current node to be a newly created node and save the start pointer position in previous node header. In case when allocated block isn't big enough for the object we trying to allocate, for instance we are creating an array of many thousand items long, we can just ask OS to map multiple blocks in a row to have a bigger sized block in the chain, which we can use in its entirety for this one array. It's important to point out that for specifically arrays or buffers Arena allocator need's special implementation to have proper use of used memory. That is due to the nature of dynamic arrays being large linear chunks of data that are constantly getting freed and allocated again, which contradicts the purpose of Arena allocator. Despite that in places where you use linked lists or any other pointer based data structure you have substantial benefits with this style of memory allocation.

Fundamentally Arena allocator is just a linked list of memory blocks (otherwise called pages in OS kernel terminology) that store last allocation pointer and next node pointer in the header. So what makes it special from GC? Technically speaking not much, it's just a data structure for organizing allocations, however, this method of organizing memory leads to a very natural for most software style of doing memory management that trivializes the process.

We can think of arena allocators as separate namespaces and lifetimes for memory allocation process, each Arena structure holds all the ownership over allocated data under a set name. If we want to allocate memory for a specific use and have it all be tracked under a known name – arena can do exactly that. This allows group allocations and has densely packed clusters of data that belong to a specific source.

The reason why it is so great is simple: we can request any amount of memory for a needed task, then once we are done, it's only one function call to clear or free all that memory.

This style of memory management allows us to think about large groups of data instead of each individual allocation, such that we don't have to spend too much effort on each individual memory object. While keeping all the control over when and how each memory space needs to be disposed.

Also when using Arena allocator we aren't forced to free memory back to the OS if we know that we might reuse it later. Each block can simply be cleared (set to 0) and only the next block pointer can be kept. When we do so we leave all the allocated memory in place and just reuse it again later when we need it. This saves quite some time because the process of freeing the memory, and allocating it back can be very wasteful, this especially holds true when working with large enough sets of data.

Also this memory allocation style naturally has performance benefits over using standard allocator, this is because of the wide range of operation of standard allocator, it can create chunks of memory of small size in very sparse locations, which causes CPU to "miss" them when copying data to CPU cache. (This term is called "cache miss"). By using data structures like lists or trees, we get a noticeable performance boost in their speed compared to allocating nodes using default allocator ("malloc" function in C, "new" keyword in `C++`).

This allocator, by design, makes lists and other similar data structures to be almost as fast as regular arrays. This was one of the discussed points about Arenas in File Pilot file manager breakdown talk by Vjekoslav Krajačić on Better Software Conference [7].

Another benefit in using Arena allocator is memory location consistency. When memory is taken from the node of the Arena allocator and we advance from the current node to a new one, that node never goes away anywhere until we free the arena. This means that creating new nodes, doesn't make an effect on all the previously allocated items. If we were to try to do the same but the source of memory would be a dynamic array, we would quickly run out of memory and needed to resize the array, by doing so the array changed its position in memory, while the previous array is either overwritten by some other process that reserved this memory after our program, or just left unused. In both cases all the previous references (pointers) to all the elements are still referencing the already freed region. This is used after free, which leads to undefined behaviour and eventually can be a "read-only write" crash.

With Arena allocator we can even (technically) "leak memory" with no consequences, if we lose a reference to allocated data, we still have it somewhere in the list of blocks tracked by Arena structure. And when they finally decide to free this Arena, all the "leaked" memory gets freed to with no consequences or risks.

Here is few example of how this approach naturally compliments almost any use case a senior software engineer can encounter:

We want to build an AST (abstract syntax tree) for a company specific scripting language, we want to allocate nodes somewhere, and keep track of them so that if we optimize out few nodes and replace them with one, we want to have no leaks or use-after free(s).

If we use a dynamic array, we run into an already mentioned problem – on resizing all the references to nodes become invalid, because junk of memory we had everything on got freed. If we use a manual approach, we have to carefully design an API that can create, delete nodes without losing a single one or we can accidentally free something we don't want, since double free(s) are also a bug that shouldn't happen in properly designed software. Another approach is creating a free list into which we append all the nodes we wanna free when we are done parsing the AST, which is a decent solution, but that requires keeping track of a separate list or array of objects, that on its own, has to be stored somewhere.

If we use Arena in this case, we save a lot of unnecessary effort, because all the nodes created with the arena are stored under that same arena, which makes them specific to the parsing process. If we don't want a node anymore, we can just unlink it and have no fear in leaking it, since after we free the Arena – it will be properly disposed of. Also we get a

benefit of almost all the nodes that are closely related, are close to each other in memory, which utilizes the cache better [8].

After we have done parsing, and already generated the instructions a virtual machine or real computer can run, we can just free the entire arena – with a single procedure call.

RESEARCH METHODS

To showcase how Garbage Collector can absolutely kill performance when misused and demonstrate how arena allocator can benefit with cache efficiency – a small benchmark will be shown. This benchmark is pretty raw, because it performs many small allocations instead of requesting large numbers of objects in chunks, but this is intentional because it can showcase flaws of using GC in Object Oriented Languages like Java or Python, because a lot of data in those languages is used in many small instances instead of storing them in buffers. Since this paper is about Arenas and GC specifically, there is no reason to spend more time explaining why the approach of doing many small allocations in OOP languages is objectively bad, so let's move on to the example.

The benchmark is simple, the program will allocate N number of nodes that hold a 32-bit integer which is the index value from the loop it was allocated in. Then each item in the list will be touched in memory to avoid compiler optimization, a sum will be computed from the loop and from iterating each item in the list. Before quitting all resources will be freed to avoid memory leaks. A memory sanitizer is used to detect illegal memory actions like: use-after-free, buffer-overflow, zero pointer dereference, etc.

Benchmark is using my own linked list macros as well as my own Arena allocator, since it's important to see how simplest implementations of all tested techniques will compare to each other. For Garbage collector implementation mkirchner's over from Github [9] is used. This GC implementation for C is relatively popular if judging by stars on the service, plus it requires no external dependencies, which perfectly ties to my implementation of data structures and algorithms goals. (written in C, portable, dependency free, simple, easy to use).

Code block 1 – Testing program in C.

```
#include "external/gc/src/log.h"
#include "external/gc/src/log.c"
#include "external/gc/src/gc.h"
#include "external/gc/src/gc.c"
#define ARENA_NODE_SIZE (4096*1024)// linux page size 64bit
#include "../hc.h"
//
typedef struct Int {
    struct Int *next, *tail, *prev;
    int value;
} Int;
//
Int* arena_int(Arena* a, int n) {
    Int* item = arena_alloc(a, sizeof(Int));
    item->value = n;
    return item;
}
//
Int* malloc_int(int n) {
    Int* item = malloc(sizeof(Int));
    memset(item, 0, sizeof(*item));
    item->value = n;
    return item;
}
//
enum { ARG_ARENA, ARG_MALLOC, ARG_GC };
```

```
#define li_next(LI) ((LI)->next)
#define li_prev(LI) ((LI)->prev)
void benchmark(int length, int opt) {
    static Arena alloc;// = {0};
    Int* list = 0;
    Int* n = 0;

    //
    for(int i = 0; i < length; i++) {
        if(opt == ARG_ARENA) {
            n = arena_int(&alloc, i);
            li_append(list, n);
        } else if (opt == ARG_MALLOC) {
            n = malloc_int(i);
            li_append(list, n);
        } else if (opt == ARG_GC) {
            n = gc_calloc(&gc, 1, sizeof(Int));
            n->value = i;
            li_append(list, n);
        }
    }
    //
    int sum = 0, precsum = 0;
    for(int i = 0; i < length; i++) {
        precsum += i;
    }
    //
    Int* iter = list;
    while(iter) {
        sum += iter->value;
        if(!li_next(iter)) break;
        iter = li_next(iter);
    }
    //
    if (opt == ARG_MALLOC) {
        li_defer(list, Int, { // available: prev, next
            free(prev);
        });
    } else if (opt == ARG_ARENA) arena_free(&alloc);
    assert(sum == precsum);
}

int main(int argc, char** argv) {
    gc_start(&gc, &argv);
    const char* arg1 = argv[1];
    const char* arg2 = argv[2];
    int opt = ARG_ARENA;
    int length = 0;
    if (argc>=3) {
        length = atoi(arg2);
    }
    if(argc>=2) {
        if(!strcmp(arg1, "arena"))
            opt = ARG_ARENA;
        else if (!strcmp(arg1, "malloc"))
            opt = ARG_MALLOC;
        else if (!strcmp(arg1, "gc"))
            opt = ARG_GC;
    }
    if (opt != ARG_GC) gc_stop(&gc);
    benchmark(length,opt);
    if (opt == ARG_GC) gc_stop(&gc);
}
```

To perform benchmark a small script inputs name of memory management strategy for program to use and number of items list needs to create. Then measures real time (kernel + userspace) spent to perform all the sum computation from list and asserts it to be equal, this means that if result isn't the same, program will crash indicating an error.

Code block 2 – Testing script in fish shell scripting language.

```
#!/usr/bin/fish
#
set iters 100 10000 1000000 10000000
set gc_iters 100 1000 10000 20000 30000
#
echo ">  Benchmarking arena"
for i in $iters
    set r (/usr/bin/time -f "%e" ./arena_vs_stock arena $i 2>&1)
    echo "$i $r" >> bm_arena.txt
end
#
echo ">  Benchmarking malloc"
for i in $iters
    set r (/usr/bin/time -f "%e" ./arena_vs_stock malloc $i 2>&1)
    echo "$i $r" >> bm_malloc.txt
end
#
echo ">  Benchmarking gc"
for i in $gc_iters
    set r (/usr/bin/time -f "%e" ./arena_vs_stock gc $i 2>&1)
    echo "$i $r" >> bm_gc.txt
end
```

After we get a set of numbers across 3 files, when merged into a plot we get a very apparent reason why garbage collector doesn't work with many small items, and why it's specifically bad for games or in the OOP when used naively, any other system with a lot of small pieces falls into this problem as well. While it takes a long time for Arena and standard C allocators to slow down due to being most O(1) time for accessing a single element of memory or freeing it, for garbage collector case it's not true, due to a need to perform a mark on intervals or (worst case scenario) every allocation – the time becomes linear, which with lookup of each variable becomes O(n^2) which grows exponentially slower as show on Figure 2. For small blocks of memory and many small elements it "explodes" execution time, even for simple actions like simply summing up numbers in a list.

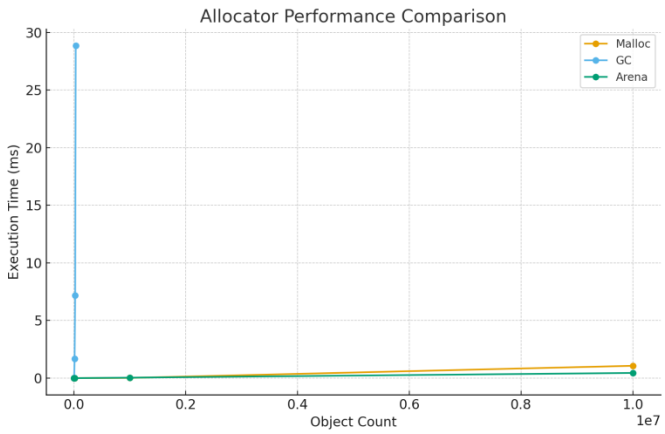

Figure 2. Comparison of execution speeds of naive allocators (manual (malloc), Arena, GC)

RESEARCH RESULT

If provided examples for Arena allocator in part about Arenas seems to be pretty selective and the benchmarks seems selective on, then you'd be surprised how often similar scenario occurs when working with most other software that is used in both Open-source and production code bases:

- processing files, strings;
- building trees, queues, graphs;
- organizing data with lists;
- creating fixed in size buffers;
- working with recursion;
- building a stack.

All mentioned data structures or problems that require memory management skills with manual methods can be trivialized with Arenas, and finely controlled in important places manually, unlike in cases with GC. Since all of the above problems are seen frequently in nearly all the software, having Arenas can prove useful in almost any project that requires any amount of control over automatic methods of memory management.

The last benefit of the Arena that has non-zero impact on making it a better memory allocation strategy is the small footprint in both source code and computation needed. Since Arena is just a small structure that holds a list of memory nodes, it isn't more difficult to implement it then a regular linked list, if default allocator is used instead of using OS specific methods – Arena allocator can be as small as little under 100 lines of code. This is a pretty respectable difference compared to a garbage collector, even the simplest one used in this paper benchmarking. The example in Code Block *3* of an arena allocator written by amuerta(author) shows how the API of list can be just a few functions and the implementation itself is small, when the value provided by the allocator is immense.

Code block 3 - Example arena allocator implementation (in C99) by amuerta [10].

```
#ifdef INCLUDE_ARENA
#ifndef __ARENA_H
#define __ARENA_H
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>
// customize block size to your desire.
#ifndef ARENA_NODE_SIZE // 2048, (4096*1024) - linux max page size
#   define ARENA_NODE_SIZE 4096 // 4kb is default linux page size.
#endif
#define ARENA_HEADER_SIZE sizeof(ArenaNode)
typedef unsigned char arena_bitmask8;
enum {
    ARENA_RESET_SIZE   = (1 << 0),
    ARENA_RESET_MEMORY = (1 << 1),
    ARENA_FREE_NODES   = (1 << 2),
};
typedef struct ArenaNode {
    size_t              allocated;
    struct ArenaNode*   next;
    char                data[];
} ArenaNode;
//
typedef struct {
    size_t      totally_allocated;
    ArenaNode*  memory;
} Arena;
```

```
//
// use these
//
void*       arena_alloc     (Arena*, size_t);
void        arena_memcpy    (Arena* a, void* data, size_t size);
void        arena_reset     (Arena*, int opt);
#define     arena_put(A, I) arena_memcpy(A, &I, sizeof(I))
#define     arena_clear(A)  arena_reset(A, ARENA_RESET_SIZE |
ARENA_RESET_MEMORY)
#define     arena_rewind(A) arena_reset(A, ARENA_RESET_SIZE)
#define     arena_free(A)   arena_reset(A, ARENA_RESET_SIZE |
ARENA_RESET_MEMORY | ARENA_FREE_NODES)
// change this one for your specific need/enviorment/taste
ArenaNode* arena_make_node(void);
//
// implementation
//
#ifdef HCH_ARENA_IMPLEMENTATION
ArenaNode* arena_make_node(void) {
    return calloc(ARENA_NODE_SIZE, 1);
}
void* arena_alloc(Arena* a, size_t size) {
    assert(ARENA_NODE_SIZE > size);
    void* ret = 0;
    ArenaNode* tail = a->memory;
    if (!a->memory) {
        a->memory = arena_make_node();
        tail = a->memory;
        goto arena_alloc_goto;
    }
    while(tail->next) tail = tail->next;
arena_alloc_goto:
    if (tail->allocated + size < (ARENA_NODE_SIZE - ARENA_HEADER_SIZE)) {
        ret = tail->data + tail->allocated;
        tail->allocated += size;
    } else {
        tail->next = arena_make_node();
        tail = tail->next;
        goto arena_alloc_goto;
    }
    return ret;
}
void arena_reset(Arena* a, int opt) {
    a->totally_allocated = 0;
    ArenaNode* node = a->memory;
    while(node) {
        ArenaNode* to_free = node;
        if (opt & ARENA_RESET_SIZE)
            node->allocated = 0;
        if (opt & ARENA_RESET_MEMORY)
            memset(node->data, 0, ARENA_NODE_SIZE - ARENA_HEADER_SIZE);
        node = node->next;
        if (opt & ARENA_FREE_NODES)
            free(to_free);
    }
}
void arena_memcpy(Arena* a, void* data, size_t size) {
    void* cell = arena_alloc(a, size);
    assert(cell && "Unexprected null, failed to allocate");
    memcpy(cell, data, size);
}
// end
#endif//HCH_ARENA_IMPLEMENTATION
#endif//__ARENA_H
#endif//INCLUDE_ARENA
```

CONCLUSIONS

Arena allocator is a great alternative to garbage collection in cases where finer control is needed, or as an overall replacement. Ability to namespace the memory comes in hand for stability and reliability of memory allocation process and comes with additional advantages such as cache efficiency that allows for better performance and smaller foot print which leads to less possibility of an error from the human implementation. Overall Arenas are a very simple and elegant way of doing memory allocations that for some reason isn't talked about or used much in interpreted languages. It would be a substantial change if such strategy was included with a dynamic language, allowing for better performance and control, while keeping ease of use that wouldn't make memory management as difficult as its portrait in most learning resources.

REFERENCES

1. Buschnick. (2010). *Garbage collection considered harmful.* https://blog.buschnick.net/2010/02/garbage-collection-considered-harmful.shtml
2. Redis. (2025). *Security advisory: CVE-2025-49844.* https://redis.io/blog/security-advisory-cve-2025-49844/
3. Oracle. (2024). *Garbage collector implementation. Java SE 24 documentation.* https://docs.oracle.com/en/java/javase/24/gctuning/garbage-collector-implementation.html
4. Anastos, M. (2019). *Unified theory of garbage collection.* Cornell CS 6120 Blog. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/unified-theory-gc/
5. Quantum Developer. (n.d.). *Demonstration of Java garbage collection flaws for game development and explanation of switching to Zig* [Video]. YouTube. https://www.youtube.com/watch?v=PxUkTxA8OWU
6. Quantum Developer. (n.d.). *CubyZ – early public alpha showcase* [Video]. YouTube. https://www.youtube.com/watch?v=jm_0nRQEn_o
7. Krajačić, V. (2025). *File Pilot: Inside the engine – BSC 2025* [Conference presentation]. YouTube. https://www.youtube.com/watch?v=bUOOaXf9qIM
8. Fleury, R. (n.d.). *Untangling lifetimes: The arena allocator*. https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator
9. mkirchner. (2019). *Simple implementation of zero-dependency garbage collector* [Source code]. GitHub. https://github.com/mkirchner/gc
10. amuerta. (2025). *handy C header(s): A personal standard library for the C programming language* [Source code]. GitHub. https://github.com/amuerta/hch

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 4 (32), 2026
ISSN 2663 – 4023

Шпак Мирослав Русланович
студент
Київський національний університет будівництва і архітектури, м. Київ, Україна
ORCID: 0009-0005-2372-8373
amurtasmail@gmail.com

Курченко Олег Анастасійович
кандидат технічних наук, доцент, кафедра програмних систем і технологій,
Київський національний університет імені Тараса Шевченка,
м. Київ, Україна
ORCID: 0000-0002-3507-2392
kurol@ukr.net

Щебланін Юрій Миколайович
кандидат технічних наук, старший науковий співробітник,
кафедра кібербезпеки та захисту інформації,
Київський національний університет імені Тараса Шевченка, м. Київ, Україна
ORCID: 0000-0002-3231-6750
shcheblanin.yurii@knu.ua

РІВЕНЬ ВРАЗЛИВОСТІ GARBAGE COLLECTION ЗА ДОПОМОГОЮ ARENA ALLOCATOR

Анотація. Розподіл пам'яті суттєво ускладнюється через поширені самокеровані фонові системи, такі як: збирання сміття, інтелектуальні вказівники та RAII. Часто ці системи та парадигми розподілу пам'яті викликають проблеми з продуктивністю та безпекою. У більшості сценаріїв (проектів) можна використовувати гідну альтернативу зазначеним парадигмам розподілу, щоб уникнути зниження продуктивності, яке збір сміття та інтелектуальні вказівники можуть накласти на створене програмне забезпечення, а також досягти кращого контролю, який допоможе уникнути можливих ризиків для безпеки між інтервалами очищення «відрізання та очищення» під час збирання сміття. Розподільник арен – це досить привабліва стратегія розподілу, яка має відносно мало згадок, незважаючи на всі зазначені переваги. Завдяки дуже лаконічній реалізації, арени можуть бути дуже зручними та зробити управління пам'яттю тривіальним, зберігаючи повний контроль у руках розробника. Арени можуть широко використовуватися майже в будь-якому проекті, пов'язаному з системами реального часу, орієнтованим на користувача програмним забезпеченням або всередині високооптимізованих систем, таких як компілятори.

Ключові слова: розподільник пам'яті; стратегія розподілу; збирання сміття пам'яті; збирач вантажу; розподільник арени; вразливість.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Buschnick. (2010). *Garbage collection considered harmful.* https://blog.buschnick.net/2010/02/garbage-collection-considered-harmful.shtml
2. Redis. (2025). *Security advisory: CVE-2025-49844.* https://redis.io/blog/security-advisory-cve-2025-49844/
3. Oracle. (2024). *Garbage collector implementation. Java SE 24 documentation.* https://docs.oracle.com/en/java/javase/24/gctuning/garbage-collector-implementation.html
4. Anastos, M. (2019). *Unified theory of garbage collection.* Cornell CS 6120 Blog. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/unified-theory-gc/
5. Quantum Developer. (n.d.). *Demonstration of Java garbage collection flaws for game development and explanation of switching to Zig* [Video]. YouTube. https://www.youtube.com/watch?v=PxUkTxA8OWU
6. Quantum Developer. (n.d.). *CubyZ – early public alpha showcase* [Video]. YouTube. https://www.youtube.com/watch?v=jm_0nRQEn_o

7. Krajačić, V. (2025). *File Pilot: Inside the engine – BSC 2025* [Conference presentation]. YouTube. https://www.youtube.com/watch?v=bUOOaXf9qIM
8. Fleury, R. (n.d.). *Untangling lifetimes: The arena allocator*. https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator
9. mkirchner. (2019). *Simple implementation of zero-dependency garbage collector* [Source code]. GitHub. https://github.com/mkirchner/gc
10. amuerta. (2025). *handy C header(s): A personal standard library for the C programming language* [Source code]. GitHub. https://github.com/amuerta/hch

Отримано редакцією журналу / Received: 18.01.26
Прорецензовано / Revised: 02.02.26
Схвалено до друку / Accepted: 26.03.26