University Hamburg
Department of Informatics
Scientific Computing Research Group

Working with Buffers

Seminar Paper

Seminar Efficient Programming in C

Christoph Brauer
0brauer@informatik.uni-hamburg.de

Supervisor : Michael Kuhn

April 15, 2013

Abstract

This paper gives an introduction to C buffers including common allocation variants and storage locations. Runtime allocation efficiency is discussed by comparison of different dynamic allocation mechanisms, and common security issues as well as according countermeasures are pointed out.

Contents

1 Introduction

The C programming language offers various buffer allocation strategies. This paper aims to impart knowledge about different allocation mechanisms in order to provide a programmer with the ability to choose the most efficient mechanism given a specific task. Additionally, common pitfalls and security weaknesses are pointed out to maximize efficiency not only in terms of ressource usage, but also in terms of security and reliability.
In order to keep the size of this paper in a convenient range the content is presented in a rather concise and compact manner. For a deeper understanding of the topics provided, this paper is supported by an according presentation [10] which in turn gives various illustrative examples and verbose explanations.

2 Introduction to C Buffers and Storage Variants

2.1 Terminology

The meaning of the term *buffer* itself heavily depends on its context, even if restricted to the domain of computer science.
To make up for this ambiguity, the term *C buffer* used throughout this paper refers to a compound and continuous area of computer memory that is assigned data of the same C datatype and allocated using the C programming language.

2.2 Buffer Variants

C Buffers can generally be allocated either statically or dynamically. Both variants differ heavily in terms of ressource usage and flexibility, so a basic understanding of those is vital in order to choose the right allocation strategy suiting the individual task.

2.2.1 Static Buffers

Static buffers feature a fixed size preset at compile time and reside in memory persistently during a program's lifetime. For these characteristics, control of static buffer allocations is given implicitly to the operating system rather than the programmer. Upon program execution, the operating system's executable file loader analyzes the program's static demands, allocates the appropriate amount of memory and finally deallocates that memory at program termination. This approach gives little to no control to the programmer and as such lacks flexibility, though it offers fast, simple and reliable operation.
Any C variable that is declared outside a function's body is by default allocated statically. To allocate a static variable inside a function body, the declaration must be prefixed by the `static` keyword. Listing 1 illustrates a typical static buffer application. Variables declared outside a function's body might as well be

prefixed by `static`, though unlike one might assume by intuition it does not specify allocation requirements but prevents a variable's symbol of being exported [12].

```c
char oneStaticBuffer[256] ;

int main ( void )
{
  static anotherStaticBuffer[256] ;
  return ( 0 ) ;
}
```

Listing 1: Static 256 Byte Buffer Example

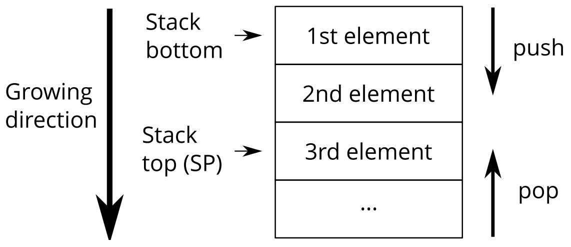
2.2.2 Dynamic Buffers

Dynamic buffers are allocated and deallocated at runtime by either implicit declarations or explicit function calls. Consequently, they offer a broad range of flexibility for they may be allocated or purged on demand and allow for adaptive size settings. Of course these major advantages do not come for free, dynamic buffers require additional ressources such as runtime management measures and special care by the programmer. Dynamic buffers may generally be allocated in any suitable memory region, though in practice only two special types, the *stack* and the *heap*, are commonly used.

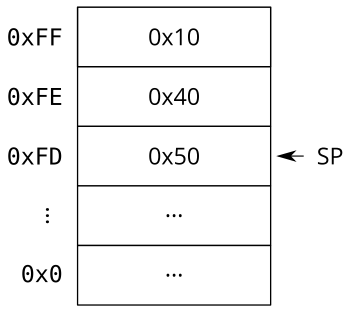
2.2.3 The Stack

The stack forms a memory area that is accessible in a LIFO (**L**ast **I**n - **F**irst **O**ut) fashion. One way to imagine the stack is to think of its elements as heavy boxes placed on top of each other. The only box directly accessible is the topmost one, so access to any other box can only be gained by removing the boxes on top of it one by one beforehand. Following this analogy, the box on the bottom is denoted *stack bottom*, the topmost box is denoted *stack top*. In terms of stack data structures, putting an element on top of the stack is called a *push* operation, removal of the topmost element is performed by a *pop* operation. Unlike the box model, memory cells can not be physically moved of course, so an index is required to keep track of the current stack position. This index is involved in any stack operation and commonly denoted as *stack pointer*, often abbreviated *SP*.
Though it might not be considered intuitive, the stack traditionally grows towards smaller addresses on many popular architectures such as i386/AMD64. In the early years of microcomputer development, a long time before virtual memory became widespread, the negative growth was used as a measure to improve memory efficiency. The stack grew downwards from the top of the physical memory whilst other data such as the heap grew upwards from the bottom. The programmer had to take special care to prevent overlapping or data corruption, and needless to say

that such a layout imposed countless errors, but at least valuable memory space could be saved.
Keeping the SP in mind, a push operation can be described as a decrement of the stack pointer followed by a write operation to the memory address pointed to by that altered SP. Accordingly, a pop operation requires a read access followed by in increment.
Both push and pop operations as well as a dedicated SP register are usually provided by CPU hardware directly.
Figure 1a illustrates the stack's principle of operation, Figure 1b gives a memory snapshot of an example stack featuring an element size of a single byte and a capacity of 256 bytes. The illustrated stack state was reached after the values 0x10, 0x20 and 0x30 were pushed on the initially empty stack, two elements were popped and finally another two values 0x40 and 0x50 got pushed.



(a) General Structure



(b) Usage Example

Figure 1: Stack Illustration

In terms of allocation demands, the stack behaves just as one might have guessed by intuition. Any element given a memory address greater than that of the SP is considered free, all others are already occupied and thus reserved. Consequently, allocating a block of memory on the stack is done by nothing but an according increment of the SP, deallocation by a decrement. In other terms, deallocation of stack data is done by abandonment.
C programs generally make extensive use of the stack as a general temporary storage. Listing 2 illustrates an example stack buffer allocation.

```
void simpleFunction ( void )
{
  char stackBuffer[1024] ;
}
```

Listing 2: Stack Buffer Allocation Example

A buffer is allocated on the stack whenever its declared inside a function's body as a function-local variable.

A C programmer is kept mostly unaware of the actual stack operations for they are implicitly created by the compiler and only appear visible in the assembler output listing. The stack is used as a shared memory between function calls, so there is only one single stack utilized throughout program execution.
Though operational details heavily differ by system architecture, compiler or calling convention, the stack is commonly used to store function return pointers, register contents and function-local variables.
The ability to store interleaved content such as program flow data and user variables makes the stack a general purpose memory, though this flexibiliy comes at the cost of major drawbacks as described in Section 4.
Whenever a C function is called, it is assigned a memory region on the stack, a so-called *stack frame*. For the stack is shared throughout program execution, nested function calls cause stack frames to be created just one after each other. Figure 2 illustrates an example stack state after a function A has called a function B. Individual stack frames are especially important for the executing of recursive functions for they allow for independent local variables across nested calls.

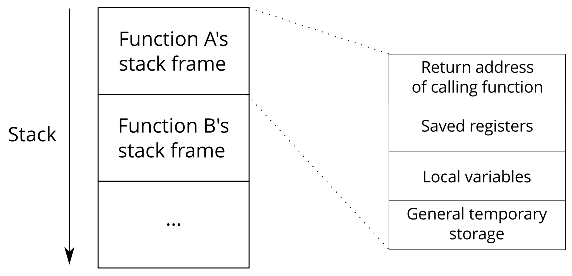


Figure 2: Stack Frame

For a function's stack frame is abandoned after the function is finished, the lifetime of local stack variables including buffers if restricted to a function's lifetime. For instance, given a stack layout as illustrated in 2, passing a pointer to function A's local data from function A to function B is unproblematic, though the other way round is errorneous for such a pointer references abandonded any most probably invalid data.
In conclusion, stack buffers provide for dynamical runtime allocation requiring minimal management effort, though the application domain is restricted due to the limited lifetime.

2.2.4 The Heap

Unlike the stack, the heap does not feature any access order conventions and as such forms a freely accessible memory region. Though heap memory itself is provided by the operating system, kernel calls are barely ever used directly in practice. One way to imagine the cooperation of kernel and user space allocators is to think of the kernel as a brewery, of an user space allocator as an innkeeper and of a function as a pub guest. The guest wants to drink a beer or maybe two, and though he could buy a whole barrel from the brewery directly, this would be a rather expensive, time consuming and wasteful approach. Instead, the guest is most likely going to buy a single beer from the innkeeper and leave the task of obtaining whole barrels to him. Additionally, this approach provides the opportunity to distribute the beer among different guests by sharing the barrel.

In terms of buffer allocations, the kernel provides for large regions of heap memory. The task of splitting the heap into smaller partitions as demanded by a program's allocation requests is performed by the user space allocator. Allocations are tracked internally to allow for memory bookkeeping and release of previously reserved memory. Once the heap space available to the user space allocator is exhausted, additional memory is requested from the operating system. Many user space allocators come with advanced features such as defragmentation measures or caching mechanisms, though these are not strictly obligatory.

One promiment example of an user space allocator is given by malloc [3, 6] shipped as part of the C standard library, but there are multitudes of different 3rd party allocators available as well. Unless a suitable allocator already exists, special tasks might as well benefit from a programmer supplying a self-written version.

3 Runtime Allocation Efficiency

As suggested in Section 2.2, buffer allocation efficiency is determined by a tradeoff between flexibility and ressource usage. In order to maximize runtime efficiency, a programmer is advised to choose an allocation mechanism featuring the least flexibility just suiting the specific task. In case neither static nor stack allocations are applicable, memory must be allocated on the heap. Though heap memory allocation generally tends to be the most ressource consuming allocation mechanism, the programmer is given the opportunity to choose an user space allocator best fitting his needs.

3.1 Malloc Allocation

The common and traditional way to allocate heap memory using C is to use the malloc allocator. It has been developed and used for many years, and though there were security concerns in the past [8], it is considered stable and generally applicable today.

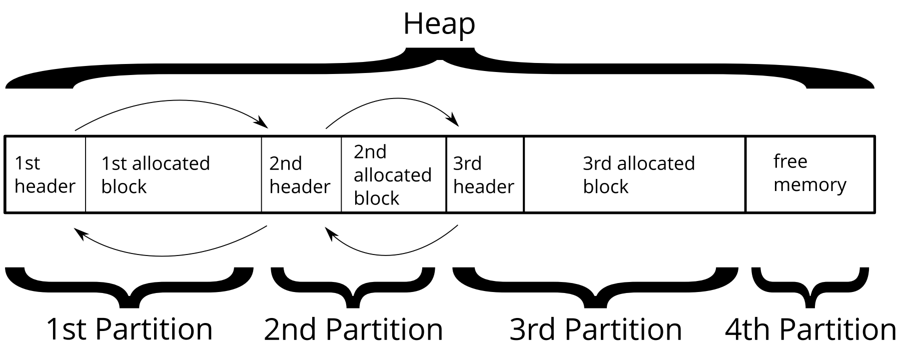


Figure 3: Malloc Heap Layout

As depicted by Figure 3, each memory block allocated by malloc is directly prefixed by an associated internal structure called *memory header*. A header contains information about the memory block next to it such as its size or availability. All headers are arranged to form a doubly-linked list, therefore operations such as insertion or removal require pointer modifications just as known from common list node operations. For each allocated block is accompanied by an according header, significant management overhead in terms of memory usage might occur given many small allocations. Popular implementations of malloc-style allocators include libc ptmalloc [6], jemalloc [11], tcmalloc [7] and dlmalloc [13], though there are many more to choose from.

3.2 Slice Allocation

Unlike malloc, a slice allocator as suggested by Jeff Bonwick [9] acts predictively for it's allocation strategy is based on the assumption that the occurence of a small memory allocation request is most probably followed by multiple similar request. The according definition of "small" varies by implementation and configuration, though less than 256 bytes are commonly considered small given that context. Allocation request that exceed this size limit are not handled by the slice allocator itself but delegated to a traditional allocator such as malloc instead.

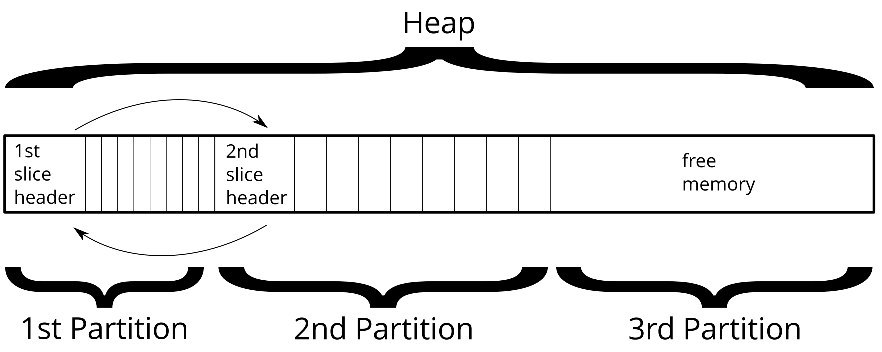


Figure 4: Slice Heap Layout

Figure 4 gives a simplified illustration of a slice allocator heap layout. A slice is made up of a slice header followed by multiple equally-sized slice blocks. The header

contains fundamental information such as the general slice status, the blocksize, the overall number of blocks and the number of occupied blocks. Slice blocks are always reserved in a consecutive fashion such that a bank of occupied blocks is followed by free blocks, if any. This layout allows for a single header controlling multiple allocated blocks and as such might noticeably decrease memory usage.
Whenever memory is requested, the slice allocator checks for an available slice such that its blocksize matches the allocation request's size. Unless a pre-existing appropriate slice is found, an according one is created, its header is updated such that the number of occupied blocks is incremented and a pointer to the block just reserved is returned.
The process of deallocation forms one of the slice allocator's major drawbacks. In case the according slice contains only one single element, the deallocation of that element causes the slice to be removed. Otherwise, the number of occupied blocks stored in the slice header is decremented and the slice is marked not available for further allocations. As mentioned before, the reduced memory usage of a single slice header controlling multiple blocks relies on the fact that a compound area of allocated blocks is followed by free blocks. A block to be deallocated is not guaranteed to lie at the end of a reserved row, so it possibly breaks the condition of the allocated blocks forming a compound area. Consequently, the positions of the remaining free blocks are left undetermined, and further allocations are prohibited.
A fully working implementation of a slice allocator is available as part of GLib [2].

3.3 Summary

Heap allocation can be performed in many different ways, and there is no strategy that can be considered generally superior. An efficiency comparison between ptmalloc and the GLib slice allocator is given in the presentation, and though the slice allocator proves that it may very well outperform malloc in certain situations, there are settings in which malloc might be the more efficient choice. For instance, a program that repeatedly executes a loop in which 2 small buffers of the same size are allocated and only one of them is freed is most probably going to perform significantly better using malloc than a slice allocator.

4 Security Concerns

Unlike many modern programming languages such as Java or Python, the C programming language does not provide any built-in capabilities to enforce buffer bounds checking. Consequently, the responsibility for proper buffer usage lies with the programmer.
Errorneous buffer usage has been repeatedly identified as a source of program malfunctions for decades [4], and though noticeable effort has been put into the development of according counter measures [14], the problem can not be considered solved yet [5]. Any buffer access that exceeds a buffer's bounds addresses a surrounding region of memory not belonging to that buffer. Such a region may contain vital data such as heap memory management structures, local stack variables or function return addresses, so especially write accesses exceeding a buffer's bounds might cause harmful consequences.
Unfortunately, the standard C library itself provides several functions such as the infamous `scanf` that work on buffers without checking proper size restrictions. One might ask how such insecure functions could ever make their way into a widespread standard code repository, and the answer is as short as simple : The standard C library was invented in the 1970s, and programmers were unaware of a future as we face it today in which computers are commonly integrated into public networks and as such popular targets of malicious attacks. Apart from the unawareness, programmers are simply humans who tend to make mistakes just like everybody else, and routines working on buffers are just as error prone as any others.
So called buffer overflow attacks provoke errernous buffer handling on intention. Most commonly, a vulnerable program is provided with selected input data to fill and finally overflow a buffer. According to the type of the attack, the program may crash, the program flow may be altered or even code provided by the attacker may be executed. Listing 2 gives an example of a typical vulnerable program.

```c
#include <stdio.h>

int main ( void )
{
    char nameBuffer[512] ;
    printf ( "Please enter your name : " ) ;
    gets ( nameBuffer ) ;
    printf ( "Hello %s !\n", nameBuffer ) ;
    return ( 0 ) ;
}
```

Listing 3: Vulnerabilty Example

At a first glance, the program does not appear to be too exciting, a name is read from the standard input, stored inside a stack buffer and finally printed on the standard output. One might be temped to assume a certain level of security for a buffer size of 512 bytes appears sufficient to store any person's real name, yet such an assumption is misleading. The given program can easily be crashed by a single shell command :

```
$ python -c "print 15000*'x'" | ./crash.elf
close failed in file object destructor:
sys.excepthook is missing
lost sys.stderr
Segmentation fault
```

Output 1: Vulnerabilty example

Numerous practical examples of different buffer overflow techniques are presented in Section 4 of the accompanying presentation, and study of those is highly recommended for a deeper understanding of potential security risks.

4.1 Countermeasures

As the enduring presence of buffer overflow bugs may suggest [1], there is no simple answer to the question about how to generally avoid buffer security issues.
Though total security can never be guaranteed, several inventions might help to at least reduce the risk of vulnerabilities. Recent compilers such as GCC 4.x offer the option to enable a built-in stack protection mechanism by validating a function's stack frame contents upon its return. Such an mechanism can not prevent stack corruptions, but at least the program is terminated and an verbose error message is given. Many modern CPUs such as those based on the AMD64 architecture provide mechanism so mark memory sections not executable. Just like the compiler stack protection, stack corruptions can not be avoided, though the risk of running malicious injected code is minimized. Last but not least modern operating systems feature an adress space layout randomization such that the address space belonging to a process is altered any time the process is run. Most exploits use fixed jump positions and thus require a known memory layout beforehand, so the risk can be minimized.
The most effective counter measure is simply given by security-aware and careful programming. Functions known to induce security risks such as the infamous standard C functions `gets`, `scanf`, `strcpy`, `strcat`, `sprintf`, `vsprintf` should generally be avoided and replaced by variants that are at least known to be less insecure such as `fgets`, `strncpy`, `strncat`, `snprintf`, `fgets`, `strlcpy` and `strlcat`. This list is by no means comprehensive, and even those functions considered secure might appear to be vulnerable in the future. New bugs are spotted every day, and security focussed internet sites should be consulted to gain latest information about security issues. Software should be tested extensively, and especially buffers should be overflown by intention to investigate potential weak spots.

5 Bibliography

References

[1] Bugtraq mailing list. `http://www.securityfocus.com`.

[2] Glib memory slice allocator. `http://developer.gnome.org/glib/2.30/glib-Memory-Slices.html`.

[3] Malloc function reference. `http://man7.org/linux/man-pages/man3/malloc.3.html`.

[4] The morris internet worm. `http://groups.csail.mit.edu/mac/classes/6.805/articles/morris-worm.html`.

[5] Multiple zero-day poc exploits threaten oracle mysql server. `http://blog.trendmicro.com/trendlabs-security-intelligence/multiple-zero-day-poc-exploits-threaten-oracle-mysql-server/`.

[6] ptmalloc - a multi-threaded malloc implementation. `http://www.malloc.de/en/`.

[7] Tcmalloc : Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[8] Advanced doug lea's malloc exploits. *Phrack*, 11(61), August 2003.

[9] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[10] Christoph Brauer. Presentation about working with buffers. `http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2012_2013/epc-1213-brauer-buffer-praesentation.pdf`, 2012.

[11] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. `http://www.canonware.com/jemalloc`, 2006.

[12] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, 1988.

[13] Doug Lea. dlmalloc. `http://g.oswego.edu/dl/html/malloc.html`.

[14] Todd C. Miller and Theo de Raadt. strlcpy and strlcat - consistent, safe, string copy and concatenation. In *USENIX Annual Technical Conference, FREENIX Track*, pages 175–178, 1999.