Memory Errors and Memory Safety: C as a Case Study

Paul C. van Oorschot version: 31 Dec 2022[1]

Abstract *We provide an accessible, self-contained explanation of memory errors as they pertain to programming languages, and the related idea of memory safety.* We use the C language as a concrete example.

For major operating system and browser vendors who make heavy use of systems languages, 65–70% or more of reported software vulnerabilities in recent years have involved *memory errors* [2]. For Microsoft, the share has remained near 70% for a dozen years, per a 2019 report [4]. In a broad study counting the number of exploits in the US National Vulnerability Database over 2013-2017, the top category was (memory) *buffer errors*, among 19 vulnerability categories [1]. Though these statistics lag by a few years, decades-old problems clearly remain with us today.

This article provides background and context for students and security novices, among others, to understand memory errors and memory safety. Definitions for these terms remain hard to find in older computer security textbooks, while discussion meaningful to practitioners is often absent from the research literature. To that end, we ground our discussion using C for context and examples of language-related security vulnerabilities involving memory errors. We plan to consider Java and Rust in a future article.

In teaching software security, we believe much can be gained by giving greater focus to the choices made in designing programming languages, rather than to malware exploits on target executables, a popular historical focus. This may help steer development teams to better programming language choices for long-term projects. A language-centered discussion of software security and memory errors may also serve students well by motivating them to learn more about comparative aspects of programming languages and their designs.

C language features and related security pitfalls

In contrast to application-level programming languages such as Java and modern scripting languages like Python, systems languages like C prioritize efficiency and programmer access to memory addresses over built-in security protections. Programming of device drivers and hardware interfaces is supported by program access to explicit memory addresses (raw pointers) and easy drop-down into assembly language or machine code.

As is well known, this comes with a downside. C and closely related C++ have historically been accomplices in a lion's share of security vulnerabilities. Certain C language features have historically been associated with programming errors underlying software vulnerabilities. To help understand *memory errors*, Sidebar-1 summarizes C language features relevant to our discussion.

C's failure to do bounds-checking on array accesses (as noted in Sidebar1) is compounded by the failure of many utilities in the C standard library (`libc`) to carry out basic error-checking. This is left as a responsibility of human programmers—who, being human, also often fail to do proper error checking. Instances of resulting mistakes produce both "normal" errors (incorrect results, run-time exceptions, program crashes) and exploitable errors (*software vulnerabilities*). For example, one type of *buffer overflow* vulnerability enables overwriting a run-time stack return address, thereby altering execution control flow.

The point to note here is that software vulnerabilities go beyond "normal" errors, and can enable remote parties to run unauthorized code (*malware*) that can take control of host machines. A novice

[1] (c)IEEE *Security & Privacy* 21(2): xx–yy, 2023, at URL-tbd

C pointer syntax, casting, basic data types, and coercion (Sidebar-1)

If `p` is a pointer, then `*p` denotes the value that `p` points to, whereas `&z` denotes the address at which the value denoted by variable `z` is stored. C variables of type *pointer* are *raw pointers* (typically 4 or 8 bytes, based on machine architecture), giving programming language access to memory addresses. While in principle pointers point to specific data types, C is a bit loose on this, e.g., `void *` (pointer to void) is *"used to hold any type of pointer but cannot itself be dereferenced"* [3]. The intent is to later manually `cast` the pointer to a specific type.

Such *type-casting* allows C programmers to convert one data type to another, including from integers to pointers. C also supports *pointer arithmetic*, e.g., you may add an integer to a pointer; the integer is implicitly scaled by the size (in bytes) of the data type of the object pointed to.

Aside from *pointers*, primitive data types in C include *integer* types of widths ranging from 8 to 64 bits: `char`, `short`, `int`, `long`, and `long long`. Integers (and floating point types) can be given modifiers `signed` or `unsigned`.

C also has a `struct` type for programmer-defined objects that group multiple (optionally different) data types, and a `union` that allows a memory area to be used by two or more different data types at different times.

A *character string* is not a basic C type, but by convention is an array of `char`, terminated by a NUL `0x00`, and referenced using a pointer to the string's first byte. Standard C library string utilities use this convention—but are a notorious source of security vulnerabilities, as discussed later in this article.

In C an *array* is a contiguous sequence of *n* data values of one type. The storage location implied by `b[i]` evaluates to `(b)+(i)` in pointer arithmetic: `(b)` denotes a base address and offset *i* is scaled by the data type (size). An array's address `&b` is the address of its first item `b[0]`. Critically, C does no bounds checking (on reads or writes) on array accesses; this is the programmer's responsibility. Out-of-bounds accesses are said to result in "undefined behavior".

To evaluate expressions, C compilers may generate code that does *coercion*, i.e., automated type conversions not requested by the programmer. This is part of type-checking system designs. For example, to ease programming of arithmetic expressions combining integer types of different width, some shorter integer data types are converted to a wider type compatible with longer integers; this is called *integer promotion*. C's specification calls for implicit coercion not only in the case of arithmetic expressions that mix integer types of various widths and signedness with floats, but also in assignment statements, function return values, and function arguments whose types do not match declared formal arguments types; a syntax error is declared if no compatible conversion exists.

might not think this concerns them directly, but their view may change when a *ransomware* attack results in all content of their laptop being rendered inaccessible.

One set of `libc` utilities is notoriously problematic: functions supporting string operations [10]. An underlying problem here is C's error-inducing convention of strings being `NUL`-terminated. The resulting security pitfalls are summarized in Sidebar-2. Warnings about these traps are given in documentation for the functions, but words in documents do not stop errors from occurring.

While what is needed is library utilities resilient to human errors, C programmers are instead expected to be familiar with countless nuances to avoid such pitfalls. As programmers continue to step into traps, security vulnerabilities creep into code, silently awaiting later exploitation. Meanwhile, various `libc` utilities are declared "unsafe" by warnings in tools, but often remain available nonetheless for backwards compatibility. The "unsafeness" involves memory errors, our main topic.

Garbage collection versus `malloc`

Garbage collection is another piece of the memory error puzzle, and involves managing heap memory—a rich source for memory errors.

When a process is scheduled to run, the OS assigns to it a *memory space* distinct from kernel memory and other processes. In C environments, this is commonly split into five regions: run-time call stack (for data related to function calls), heap (for dynamically allocated objects), text segment (for code), data segment (initialized global data), and the BSS area, short for *block started by symbol* (uninitialized global data).

Often in systems languages, heap memory is *manually allocated* by programmatically requesting blocks of the desired size (*chunks*). In C, calls to `malloc()` return a pointer `p` to a chunk of uninitialized heap memory, or a `NULL` pointer if the request cannot be filled; the related `calloc()` returns a zero-initialized chunk. Memory no longer needed is released by calling `free(p)`; if not, the memory is effectively lost (called a *memory leak*), inducing out-of-memory errors over time.

Manual memory management is avoided in many higher-level languages (e.g., Java) through a process called *garbage collection* (GC), whereby memory chunks no longer used or reachable are automatically reclaimed and returned to a system *free pool* of available memory chunks. Any of various *storage allocator* approaches are used to manage this heap memory, often arranging the *free*

C strings, string utilities, and related buffer overflows (Sidebar-2)

`strcpy(char *dst, char *src)` is a main string-copy utility in C. It copies whatever string is found at the `src` address to the address indicated by `dst`, byte-by-byte, ending once a `NUL` byte is found and copied. If fewer bytes were allocated for the destination buffer than present in this source string, that does not matter—the copying blindly continues until a `NUL` byte is found. This overwrites memory beyond the end of the `dst` buffer, continuing into the adjacent data structure or executable code at the next higher memory address, and the one after that and so on. Eventually a `NUL` byte will be copied. Not all such instances are exploitable; by this we mean, it might "only" trigger an access violation or system crash.

While such an overrun can arise if the `dst` buffer is shorter than the `src` buffer, it can also occur even when the `dst` data structure is the same size or larger than that of `src`, in the case that the `src` buffer itself contains no ending `NUL` byte. This can result in a different problem at the `src` end: reading of non-`NUL` bytes beyond the end of the `src` buffer may result in an *information leak* exposing potentially sensitive data (e.g., secret keys) from memory beyond the intended `src` buffer.

`gets()` and `puts()`, the get-string and put-string functions, raise similar issues. Another example is the string concatenation function, `strcat(char *dst, char *t)`; it concatenates the string at `t` to the end of that found at `dst`, blindly assuming that `dst` has sufficient space [3].

Yet another is `strncpy()`, perhaps viewed as a safe alternative to `strcpy()`. It takes a third parameter n, the maximum number of bytes to copy, stopping after either the first `NUL` or n bytes, whichever is first. This enables a different error: the resulting destination string may end up without a terminating `NUL`. In that case, a later memory error will almost surely occur as later code expects a conventional `NUL`-terminated string.

`strncat()` is a concatenation alternative with a third parameter n, here specifying the maximum number of non-`NUL` bytes concatenated to the end of the `dst` string, but a terminating `NUL` is always inserted—extending the `dst` string by possibly $n+1$ bytes. This inconsistency with n's use in `strncpy()` induces so-called *off-by-one* memory errors, corrupting single bytes.

`strlen(s)` is another off-by-one trap. It returns the number of bytes in its string argument, excluding the ending `NUL` byte—so one byte more than the return value is needed to store the string.

pool in a double-linked list ordered by size, and occasionally reorganized.

GC has advantages beyond relieving programmers from the burden of allocating and deallocating memory: it eliminates *temporal memory errors* (as explained shortly) and can significantly reduce memory leaks. However, disadvantages include significant processor time costs, and unpredictable process delays when the GC routines engage to collect and consolidate free memory. Basic GC approaches are also unsuitable in time-critical applications that must meet strict timeliness constraints.

Data types and type checking

As we build towards defining *memory errors* and *memory safety*, it helps to recall the main ideas of data types, type systems, and type checking [5].

Why are *data types* needed in programming languages? Many operations—even in simple arithmetic expressions, and assigning a value to a variable—take inputs whose values are expected to be from specific domains and produce outputs with expected characteristics, again with values falling in an implied domain [9]. Data types are used to specify and convey these expectations, and every programming language has some form of *type system* that defines built-in types, allows creation of user-defined types, and dictates rules governing what is required and enforced related to these types.

Data types are associated not only with data structures, but also functions and operators (including unary and binary arithmetic and logical operators) via the allowed inputs and outputs—which must have expected types.

How to define *data type* in greater detail is somewhat harder. The important aspects are that types convey useful details about an object's properties and semantics, and of help in identifying mismatches between objects and their expected operators or uses. In essence, data types convey the operations or methods allowed on data values—and of special interest to us is catching errors having possible security implications. Data types also convey information to the system about how data values are represented (e.g., storage formats); this may be intentionally hidden from application programmers in an attempt to simplify their task, or to abstract platform-specific representations.

Based on a type system's types and rules, *type checking* is the process of detecting *domain incompatibilities*—improper or unexpected uses of objects—at compile time (*static* type checking), run time (*dynamic* type checking), or both. Not all incompatibilities can be detected at compile time, as not all values are known. Out-of-bounds array accesses when element indexes are dynamic inputs, and division by zero, are examples of errors that cannot be caught at compile time; examples that can be are multiplication or exponentiation of strings and boolean values.

Requiring a programmer to specify the data type of every single object and value is tedious. Thus to varying degrees, programming languages relax the requirement of explicit *type declarations*, some requiring few if any. The language processor then uses *type inference* to assign a data type (hopefully as intended by the programmer) based on context and the rules of the language. Of course, this might result in a mismatch with programmer's expectations, and a syntactically acceptable program having semantic errors. Thus there is a language design tradeoff: strict requirements of type declarations impose heavy burdens, while being lax risks undetected mismatches between programmer intent and a language processor's actions.

When the operands of an operation or arguments of a function are incompatible by strict rules, a programming language could simply declare an error. Another option is to generate code to automatically convert one or more values to a compatible type, in cases where this make sense such as addition of integers and floating point numbers. This is called *coercion* (implicit type conversion).

There are different sets of rules by which this might be done—for example, consider the possibilities for adding an integer and a float, `(2 + 3.0)`. A natural choice is to convert the integer

"upward" to a float. Along these lines, well-defined (albeit not always well understood) *conversion hierarchies* have emerged. These resolve some, but not all, attempted operations—e.g., most languages disallow "adding" an integer to a string (though some treat it as concatenation).

Implicit type conversions can be more complicated than they first appear, even for integers and floats—consider, for example, that C has numerous widths for integers alone, as well as `signed` vs. `unsigned` (and also `char`). Languages that are overly eager in carrying out coercion enable programming errors—some exploitable—when unanticipated by programmers, and may signal what is called *weak typing*.

Static type checking may result in compile-time errors, or in generating code for run-time coercion or run-time type checking. If a compile-time check can rule out a type incompatibility, then this single check both avoids a run-time test at each instance the relevant code line runs, and catches errors during development versus during dynamic testing or post-deployment. Static type declarations themselves also often help developers understand programs.

C is weakly-typed

C is *statically typed* (variable types are declared) and has *static type-checking* (expressions and function parameters are type-checked at compile time). However, note that C supports or allows:

- a wide range of implicit type conversions;
- raw pointers, pointer arithmetic, and converting pointers to one type to pointers to another;
- various code sequences noted by the specification to result in *undefined behavior*;
- no bounds-checking on memory accesses via pointers, although compiler-related tools may offer special options or warnings;
- bypassing the type system by the `union` construct, and manual casting (type conversion) including between pointer types and integers, and from a function pointer to another whose function signature differs (e.g., in return value type, argument types, or number of arguments).

While Kernighan and Ritchie [3] state that every (non-void) C pointer points to an object of some fixed type, C's design takes no responsibility for enforcing this, as they also state, e.g., that the programmer is responsible for tracking which type is currently stored in a `union`. Programmers are likewise responsible for tracking the data type of a pointer's referent, as is clear from the flexibility allowed in casting pointer types, pointer arithmetic, and array indexing. Thus, although C is a *typed* language and variables must be declared (with a type name) before use, it is loose (relative to other languages) in ensuring that object types are compatible with uses.

Based on these observations, C is said to be *weakly-typed*. Its design and language processors do not reliably guarantee domain compatibility in the use of objects, and no run-time type-related support addresses compile-time deficiencies. The opposite is a *strongly-typed* language.

Memory error categories and memory safety

We now consider categories of security problems enabled by language features.

A wide variety of errors involving improper memory access, on both writes and reads—called *memory errors*—are enabled by dereferencing of invalid pointers, and failure to check that array accesses are within bounds [7], [8]. Some, but not all, result in error conditions raised by the hardware (CPU); in some cases, errors enable unauthorized code to be executed before (or without) the vulnerable process being terminated, or the operating system crashing.

Consider first a few possible outcomes of *writing* to an unintended address, e.g., beyond the bounds of a referenced array. Perhaps the value written is controlled by a malicious program input.

a) The value overwritten is a *code pointer* (stack return address or function pointer). This will alter later execution paths when the code pointer is loaded into the CPU instruction pointer, breaking the program's *control-flow integrity*.

b) The memory overwritten holds program data other than code pointers, e.g., data variables and pointers to them (*data pointers*). This breaks program *data integrity*. It may also indirectly alter execution paths that depend on the altered values.

c) The memory overwritten is code that will be executed later. This directly breaks *code integrity*. A common tactic involves *shellcode*, where an attacker crafts special code as malicious program input, aiming to execute this code of their choosing on a target machine.

Consider next two outcomes of errors associated with *read* access, which may lead to information leaks (disclosure of sensitive data).

d) Data is read from an *uninitialized object*, part of whose associated memory retains values from when the same memory was used for an earlier object. (C does not require initialization of automatic variables; static vars are set to 0 or NULL. Reading from an uninitialized variable is said to result in *undefined behavior*.)

e) Data is read from an arbitrary address within the process' address space, unintended or unanticipated by a benign programmer. A severe example was the 2014 *Heartbleed* incident, where a function in the OpenSSL library failed to check array bounds.

Distinct from these outcomes, we now consider three categories of memory errors [7]. The first category is *spatial safety errors* (spatial refers to a memory range), including two cases:

1(i) memory access (read or write) that involves a pointer to one object, but results in access to memory outside the range allocated for that object. For write access, this corrupts a separate object. The first object could be an array whose elements are accessed using a base pointer and offset; the error equates to a failure to bounds-check. In the buffer overflow case, bytes are written continuously beyond the buffer's end, spilling into objects at higher addresses.

1(ii) dereferencing a *wild pointer*. We define a *wild pointer* as any pointer whose use would result in undefined behavior per the language specification (e.g., in C, for uninitialized and NULL pointers, among others). NULL pointer dereferencing is known to be exploitable in some cases (vs. simply causing an access violation). NULL, often represented `0x0..0`, might map to kernel memory through the virtual address translation implemented by the OS and hardware.

The second category is *temporal safety errors* (temporal refers to time), with two cases both involving use of a *dangling pointer*:

2(i) *use-after-free* error. This involves dereferencing a *dangling pointer*, i.e., a pointer to an object in memory that has already been deallocated. The referent is an *invalid object*, and using any reference to it is an error. The object referred to might be in heap memory, or a local variable on the stack call frame (and referred to after its memory is deallocated by a function return).

2(ii) *double-free* error. This involves freeing an already-freed object, thus passing the deallocator a dangling pointer. Deallocation puts a memory chunk into the *free pool*; doing so twice may corrupt internal allocator data structures (if a free pool chunk is freed a second time), or may create a separate dangling pointer (if the chunk was already re-allocated to a new object).

A third category of memory errors (also resulting in *undefined behavior* in C) involves:

3) reading from *uninitialized variables*. Here we exclude dereferencing wild pointers, viewing that as a spatial error—but include the case of an uninitialized pointer whose value is read (e.g., to assign to a second variable) but not dereferenced. (If and when the second variable is later dereferenced while holding a wild pointer, we view that as a spatial error.)

Aside from the third category, memory errors involve dereferencing or using an invalid pointer.

Memory safety levels L1 to L4

We can now define *memory safety*, or rather, four levels useful to differentiate classes of memory errors that a language's design and core support tools (compiler, run-time) may address.

L1: *fundamental memory safety*. Level 1 aims to eliminate *spatial safety errors* and *temporal safety errors*, the most serious categories of memory errors and pointer-based security issues. L1 safety may be viewed as guarding memory associated with an object through checks on memory bounds (low, high), plus a flag indicating whether the object remains *valid* (i.e., both its memory remains allocated and the object remains in scope).

L2: *clean memory safety*. Level 2 aims to eliminate both information leaks and undefined behavior related to uninitialized variables.

Further levels are associated with mitigating two other common classes of memory errors.

L3: *memory leaks*. Level 3 aims to eliminate memory leaks. These can crash programs or the OS, and can be viewed as security-related in that they may enable denial-of-service attacks.

L4: *data races*. Level 4 aims to eliminate security issues and unpredictable outcomes due to *data races*, which can arise in the case of concurrent reads and writes to shared memory, e.g., if one execution thread changes a data value while another is using it. This and related concurrency issues fall in the broader context of *thread safety*.

Other categories of memory-related errors exist. Two of these overlap L1 and L2: errors involving exploitable *format strings* (e.g., user-defined or user-controllable formats in C's `printf` family of formatted-output functions), and *variadic functions*, taking a variable number of arguments [7]. Another protection category involves cleartext secret keys and passwords in memory (some systems offer support to store these encrypted while in memory, aside from instants of actual use).

Memory safety levels L1–L4 offer a useful starting point for assessing and comparing programming languages with respect to their features that may help prevent memory errors, and thereby improve software security. From our discussion, it is easy to see that C has deficiencies in each of the four levels. Languages such as Java and Rust, among many others, fare better in most areas.

Cifuentes and Bierman [1] suggest that mainstream programming languages leave much room for improvement, in that all fail to provide features to preclude prominent, known categories of software vulnerabilities (from a list of categories well beyond those considered herein). But which of these categories can be effectively addressed via programming language design remains unclear.

Overall, we view memory errors as violations of the expectations, set by a language specification and its abstractions, about how programs should interact with memory [6]. Here we should also emphasize the separation of a language specification from its implementations (thanks to F. Piessens for this reminder). For example, the C specification declares various instances of memory errors to result in "undefined behavior", and does not take responsibility for ensuring that all memory accesses are proper (consequently, C is declared *memory-unsafe*); the degree to which these memory errors result in security vulnerabilities can vary greatly across compilers and runtime environments.

References

[1] C. Cifuentes, G. Bierman. What is a secure programming language? Summit on Advances in Programming Languages (SNAPL), 2019.

[2] A. Gaynor. Introduction to memory unsafety for VPs of engineering. Aug 12, 2019. https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/.

[3] B. Kernighan, D. Ritchie. *The C Programming Language (Second Edition)*, Prentice Hall, 1988.

[4] M. Miller (Microsoft). Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat Israel Conference, slides 1–24, 7 Feb 2019. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[5] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[6] F. Piessens. Software Security Knowledge Area (CyBOK, Version 1.0.1), July 2021. https://www.cybok.org/media/downloads/Software_Security_v1.0.1.pdf

[7] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, M. Franz. SoK: Sanitizing for Security. IEEE Symp. Security and Privacy, pp.1275-1295, 2019.

[8] L. Szekeres, M. Payer, T. Wei, D. Song. SoK: Eternal war in memory. IEEE Symp. Security and Privacy, pp.48-62, 2013.

[9] R.D. Tennent. *Principles of Programming Languages*. Prentice-Hall International, 1981.

[10] D.A. Wagner, J.S. Foster, E.A. Brewer, A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. NDSS, 2000.