Memory Errors and Memory Safety: A Look at Java and Rust

Paul C. van Oorschot **version: 4 Mar 2023**[1]

Abstract. We consider *memory errors* and *memory safety* in the cases of the Java and Rust programming languages. We also give a view of how *type safety* fits in.

Our previous article [13] discussed memory errors, with a focus on the C programming language. We distinguished four levels of memory safety, with L1 (*fundamental memory safety*) protecting against spatial and temporal errors, L2 (*clean memory safety*) eliminating uninitialized variables, and further levels L3 to mitigate *memory leaks*, and L4 supporting avoidance of *data races*.

We now continue, first considering *type safety*, and then returning to memory errors with a few more comments on C, and main focus on features in the Java and Rust programming languages.

Type safety and type confusion

C is often said to be *weakly-typed*. Despite being a *typed* language (the type of a variable must be declared before use), programs are not fully *type-checked*. A program that compiles without errors may still yield *undefined behavior* [2], by using constructs specifically identified by C's specification (thus implicitly discouraged) as allowing a compiler to do anything and still claim compliance.

Such undefined behavior is inconsistent with security. We want language designs, processors, and run-time suppport aiming to ensure variables and expressions have types consistent with operations they are used in, and to prevent memory assigned to values of one type being used as another.

When a language processor or run-time system (or program) has inconsistent views of an object's type, or believes an object has type X when last stored out as an incompatible type Y, we say there is *type confusion*. This may arise from a semantic program error (for a language that does not rule out inconsistencies), or a tool implementation error (e.g., failing to detect an inconsistency that a language aims to rule out). Type confusion may lead to behavior unexpected by the programmer.

In *weakly-typed* languages, a given instance of type confusion may be unintended (and undetected if syntactically allowed). In the case that a programmer intentionally refers to the same object (region of memory) as having two incompatible types, it may be called *type punning* (from *pun*, a play on words, where a word or phrase has two meanings). For example, C `union` types (untagged unions) enable type punning, effectively bypassing C's type system. Reading a value out from a union member as type A, that was last stored into as type B, should generally be expected to result in an improper interpretation of the value.

It is desirable that the design of a programming language type system assign distinct types to objects with distinct uses or semantic properties. For example: the type for pointers to data (*data pointers*) should be distinct from function pointers, and the type for integers distinct from pointers—e.g., because a pointer to a data object in memory plays an addressing role (locating and accessing data), whereas regular integers are used for arithmetic expressions.

Now, consider the term *type-safe language*, sometimes shortened to *safe language*. Characterizations of this term by experts include the following [2, 15].

- *A safe language guarantees the integrity of the abstractions that it provides.* This sets the expectation that abstractions defined through the language are enforced at run time (e.g., boundaries of objects, allowed methods to access array elements).

- *A safe language has a specification that fully defines program behavior, ruling out undefined behavior.* The language takes responsibility for ensuring that programs behave predictably (versus making programmers responsible for the consequences of unsanctioned constructs).

[1] (c)IEEE *Security & Privacy* 21(3): xx–yy, 2023, at URL-tbd

- *A safe language precludes untrapped errors at run time.* The language promises that any error not detected statically will be trapped immediately upon occurrence at run time, and that there can be no undetected run-time errors (e.g., which might cause unexpected behavior later).

While the above are characteristics rather than a formal definition, *type safety* is given precise definitions in type system literature, (Some experts also use the term *strongly-typed* interchangeably with *type-safe*, but others give it a separate meaning; we thus prefer to avoid it.) Type-safe languages often rely on both compile-time and run-time checks to find targeted errors, specific to the language and the goals of its type system.

These goals might not align with intuitive expectations in all cases, but in practice, type safety often goes hand-in-hand with desired memory safety properties. Practitioners want memory safety, and type-safe programming languages typically deliver at least temporal and spatial memory safety, as part of eliminating undefined behavior.

Relating language choice and software security

Garbage collection is used by most languages that offer memory safety guarantees [15], as a common means to avoid errors related to manual memory management, including memory leaks, spatial errors and temporal errors. As should now be clear, the language used to write a program has a huge impact on the resulting software security—as some languages eliminate entire categories of errors.

However, language choice alone is not a full solution to avoiding memory errors related to systems languages like C++ and C. These are often also called compiled or *native languages*, in contrast to high-level languages that run in virtual machines or interpreters [16]. The challenge is that "memory-safe" high-level languages may themselves have supporting components or use run-time libraries that are written in native languages—e.g., to interface to hardware, or for performance or historical reasons (legacy code, backward compatibility).

As a sign of progress, some categories of simple buffer overflows are becoming harder to exploit in more recent products, due to not only improved development tools that help avoid them, but also effective run-time defenses. For example, as of 2019, Microsoft reported that stack buffer overflows had almost disappeared among their vulnerabilities [11].

However, languages themselves cannot eliminate all security problems related to software. As one example, definitions of memory safety and type safety are silent on language-related categories of errors such as *integer overflows* and *underflows* (below). As another, language-based software security is orthogonal to *social engineering* ("download and install this cool software [please ignore the malware hidden within it]").

Furthermore, neither memory safety nor type safety address security issues related to lack of *input validation*—e.g., preventing a `char` string <`script`> tag, embedded in user input to a web discussion forum, from being executed as JavaScript. This type of input validation may be viewed as a higher-level type checking issue above programming languages. For discussion of the failure of mainstream programming languages to provide features that protect against this, related *injection errors* and other major categories of software vulnerabilities, see Cifuentes and Bierman [3].

Integer overflows (*underflows*) are known to indirectly enable a wide variety of exploits. For example in C, if a 16-bit `unsigned int x` holds value $2^{16}-1=65535$, `x+1` yields 0. We commonly call this an integer overflow, but per C's specification, arithmetic overflow "never occurs" for an `unsigned int`—instead, for an n-bit width, the result is computed in arithmetic modulo 2^n and considered correct (with no error flagged). In contrast, for an 8-bit `signed char` $x=127$, C says that adding 1 yields undefined behavior (most implementations yield $x=-128$ and proceed).

Integer overflow/underflow, and consequences of coercion (e.g., integer width conversions, truncations and extensions) can lead to a broad class of errors called *integer-based vulnerabilities*, which are not themselves but can indirectly contribute to memory errors (e.g., via integers used in branching conditions, as loop counters, or array access indexes). Surprises often result from mixing signed and unsigned integers. For example in C, comparing signed and unsigned `int` values results in *promotion* (a coercion) of the signed `int` to an unsigned of same width. This converts negative values to positive, often unexpected by developers; compilers may or may not warn about this. Thus the general advice across programming languages: avoid mixing signed and unsigned data types.

A closer look at Java

We now look at Java as a language example, with focus on security-related aspects including data typing and memory safety. Java is distinguished in that its programs are compiled to *bytecode*, which is then loaded and run on the Java Virtual Machine (JVM).

Java was designed to allow substantial type checking at compile time, but some checks must be done dynamically, e.g., for objects whose types are not statically known. Thus it is statically *typed* (variables must declared before use), but not entirely a statically-*checked* language.

References are used in Java, not raw pointers. This is common for high-level languages that use *garbage collection*, as Java does. A Java variable for a (non-primitive) object stores not a value, but a *reference* that provides access to a value on the heap. For example, for `rec = new Record();` the identifier `rec` may allow access to `rec.firstfield`, but no memory address can be derived by the programmer from `rec`. This rules out program-based manipulation of addresses.

The internal implementation of a Java reference to an object (e.g., a growable string or vector) nonetheless involves a pointer to the value (as well as metadata enabling validity checks). But Java's design eliminates C's explicit pointer dereference (`*`) and address (`&`) operators.

Java allows setting a reference to empty (`rec = null`), implying no object present. The variable itself then stores a semantic value denoting `null`. Trying to dereference a null reference, as in `rec.firstfield`, throws an exception—albeit called `NullPointerException` for historical reasons. Thus Java's implementation of a *reference* does not simply try to follow a raw pointer as would be done in C; in contrast, it involves semantic checks, and bounds-checking.

If `aa` and `bb` are variables for objects of the same type in Java, then `bb = aa` does not result in a copy of object `aa` being created for `bb`; instead, `bb` becomes a reference to the same object as `aa`. We say that `aa` and `bb` are *aliases* for the same object; modifying the value of one changes both.

Type-casting in Java

In Java, *casting* creates a reference to an object having a type compatible with the original (the original object is unchanged). Here to be *compatible* requires an existing relationship within Java's object inheritance hierarchy. *Valid* casts include explicit *downcasts* from a parent *superclass* type to a child *subclass*, and *upcasts* from a child class to its parent class (*supertype*). Primitives can be downcast (*narrowcast*) to narrower types, e.g., `int` to `short`, or `float` to `int`; the expression `(int) doublefloatx` could be assigned to an `int` variable, narrowing to a value of 10 if variable `doublefloatx` had value 10.8. The opposite, a *widecast*, is done implicitly (automatically) by the compiler when needed; however narrowing requires explicit casting to avoid an error.

While some casting errors can be statically detected, if a type is unknown at compile time, compatibility checks are done at run time. Run-time cast errors result in a `ClassCastException` being thrown. *Run-time type information* (below) is thus needed for type-checking dynamic casts—and for expressions involving the binary operator `instanceof`, often used to avoid this exception. For example, `(obj instanceof class)` returns a boolean, allowing control-flow decisions based on a run-time test of an object's type against a statically known type `(class)`.

Recall that defining a Java *class* creates a new Java *type*, which variables can be declared to have, thereby granting access to methods within the class. On compilation, the resulting *class file* contains information loaded later by the JVM. From this the JVM builds a dictionary of *run-time type information* [17], and creates an object of class `java.lang.Class` for each type loaded (providing class metadata and *methods*). To use this type information in support of runtime type-checking (such as for dynamic casts), object instances in the JVM are associated with the dictionary type information for the object's class. This might be done by a pointer from the object instance to the dictionary type information.

Java is generally said to be *memory safe*. Our L1 is delivered by Java's design of references (no program access to explicit addresses), type checking and casting rules, avoiding temporal errors by garbage collection, and avoiding spatial errors by checking object bounds through compile-time and (when not statically reasoned to be always safe) run-time bounds-checks. L2 is achieved by flagging use of uninitialized local variables as a compile-time error, and assigning default values to uninitialized class variables (and fields therein): 0 or `false` for primitive types and `NULL` for objects. For L3, memory leaks can be substantially reduced by garbage collection. While Java remains susceptible to programming errors related to *integer overflow*, its specification does require that the resulting behavior be as expected from two's complement arithmetic. (In contrast in C, overflow of anything other than `unsigned` integral types officially yields undefined behavior.)

For discussion of Java support to prevent *data races* (L4), including through synchronized access (e.g., `synchronized` statements and `synchronized` methods), see Eck [4]. For high-level explanations on Java type-checking and type confusion, see McGraw and Felten [10], and for a long-term review of Java security vulnerabilities, see Holzinger et al. [8].

Rust overview and motivation

Awareness of C's memory safety failures and the security implications continues to grow. The Rust programming language is arguably the most serious contender as a systems-level alternative to C, i.e., a lower-level language suitable for operating systems and browsers and interfacing to hardware. Here we briefly introduce a selection of Rust's design features related to memory safety and security, and encourage pursuit of further details independently.

While the importance of security in programming languages has been known to experts for over 35 years—even before the 1988 Internet worm incident—it is now reaching wider audiences. Recent efforts to deliver this message include Gaynor's 2019 pitch [7] encouraging VPs of Engineering to choose languages other than C/C++, and a high-level overview of memory safety from the US National Security Agency in November 2022 [12]. A month earlier, Linus Torvalds approved direct support for Rust kernel code in Linux 6.1. Arguments in favor of Rust over C for introductory OS courses date back as early as 2013 [5].

We now begin a short tour of Rust design features that support security. As a central aspect, Rust developers must be aware of the location of memory associated with an object (heap, stack, or a data segment). Heap memory is of special concern, as heap objects internally involve raw pointers pointing to values. Rust's design is such that no object is pointed to by more than one pointer (at any one time) having the capability to modify the object's value.

Such write-capable *aliases* are historically a root cause of security vulnerabilities in C, and aliases in general complicate static detection of various dangerous practices. Rust's design enables detailed static analysis, and provides significant features for handling errors at run time. Together, this addresses many of the memory errors that we have discussed.

Mutability and ownership

Mutability is a key concept. By default, the value of a Rust variable cannot be changed once assigned. To do so requires an explicit declaration of the variable as *mutable*, using keyword `mut`:

```
1: let mut i = 7;     // we say this binds the integer value 7 to variable i
2: i = i + 1;         // update allowed, because the variable is mutable
3: let j = i;         // type inference used here (type declaration optional here)
```

After line 1, `i` is said to *own* the value `7`. At line 3, a copy of value `8` is made and that copy is bound to variable `j`. Both `i` and `j` have value `8`, stored as separate copies on the run-time stack.

If instead of integer variables `i` and `j` we now consider variables `u`, `v` of type `String` (below), and `u` had string value `"STRing1"`, then `let v = u;` would *not* result in a copy of the string being made. This is because dynamic strings are stored in heap memory, and by default Rust does not copy heap data (sometimes called *deep copying*). Instead, value `"STRing1"` is said to be *moved* from `u` to `v`, meaning that *ownership* (as tracked by the compiler) of the original copy of the value transfers to `v`. The value is no longer accessible through `u`; attempted access yields a compiler error.

The rules on *owning* and *moving* values apply to all assignments—not only in explicit assignment statements, but also in function calls, both when variables as instances of actual arguments are converted to formal parameters, and in passing return values back to calling functions.

Passing a variable of type `String` to a function *moves* the string value (and its ownership) to the scope of the function. However, passing a primitive-type variable (such as `i:u32`, unsigned 32-bit) to a function does not move ownership of its value; instead, a copy is made, based on the design decision that a `u32` (as a primitive of known size) can be efficiently *shallow-copied* as stack data.

Ownership rule #1 in Rust is: *Each value has a single owner object at any one time.* The owner is the only entity with authority to alter the value. Thus, a single owner has *write* privileges.

Scope rule #1 is: *When an object goes out of scope, associated dynamic memory is freed.* (Code to call a `drop` function is automatically compiled in.) As a baseline, the scope of a non-global object is the function or block it is in; ownership can be transferred out of a block through a return value. Rust tightens an object's scope (*lifetime*) to its minimum span of actual use within a block (e.g., rather than the end of a block or function); in some cases this avoids otherwise violating *Ownership rule #2*, below. Beyond an object's lifetime, neither the object nor its value is *valid* for access; compiler errors (on attempts to access invalid objects) clarify valid spans.

A variable's scope may be refined by blocks delimited by braces ({ }) and by explicit annotation (using optional scope labels in variable declarations). While these scoping rules at first appear to add time and complexity burdens on programmers, they enable detailed compile-time type checking that helps prevent memory errors and related security vulnerabilities. The win is long-term, in the form of fewer bugs, as a result of the safety guarantees that the compiler can deliver.

References and borrowing in Rust

The syntax `&i` denotes a *reference* to variable `i` (not its value). The reference can be used, e.g., to assign to another variable or as a *pass-by-reference* argument to a function (vs. passing a copy of the value, *pass-by-value*). As an example (`&i32` declares a reference to a 32-bit int):

```
let b: &i32 = &a;    // b is a reference to a
```

Rust aims to guarantee that throughout the lifetime of a reference, it always refers (points) to valid values of a given type. References to non-primitive types are *smart pointers* (as discussed shortly).

Like other variables, references are immutable by default; to modify a value through a reference, it must be a *mutable reference* (`&mut`) and the object to be modified must also be declared `mut`.

`&` is called the *borrow* operator. An `&` reference allows reading, and is said to *borrow* a value. A mutable borrow (`&mut`) also allows write access, and *moves* a value's ownership. A borrow's read access privilege ends (is returned) when the variable's lifetime (scope) ends. This is tracked by the compiler's *borrow checker*, which enforces our second ownership rule.

Ownership rule #2: Rust allows only one mutable reference to a value (and no immutable references in this case), or multiple immutable references (allowing read-only access).

Some data types are fixed-length once declared (e.g., *array* and *tuple* types). Others are variable-length, e.g., `String`, and *vector* type `Vec<T>` for growable collections of a fixed basic type `T` (say, `u32`). Accessing an invalid collection element, e.g., the ninth element of a vector with only elements 0..7, is caught as an error. For example, a vector element access may be attempted by: `&myvec[8]`, or `myvec.get(8)`. The first would trigger a run-time *panic* error (program termination). The second would return a semantic value `None`, enabling programmatic error-handling. Notably, in neither case do (L1 spatial) memory errors occur.

Option types, NULL pointers and strings

To eliminate NULL pointers in regular code, Rust has a generic type `Option<T>`, with type parameter `T`. For references to objects `x` of type `T`, `Option<T>` can be assigned values `Some(x)` for the non-NULL case, or `None` for the NULL case (the absence of a value). In the case of `Some(x)`, the value `x` that is *wrapped* in the `Some` variant can be accessed using `.unwrap()`. Executing unwrap on `None` results in a runtime *panic* (typically, program termination accompanied by an error message).

This leads to programs being written to explicitly handle both cases, with the `Some` and `None` cases distinguished using common Rust constructs and methods (which we do not discuss here). This replaces *ad hoc* (often missing) NULL pointer checks in C code. While C programmers and novice *Rustaceans* may initially view this arrangement as tedious, the tradeoff is short-term attention to detail to avoid often large costs later, to fix hidden bugs and their consequences.

The `Option` type also appears in one of the built-in method categories available for custom handling of integer overflow. (Arithmetic overflow results in defined behavior in Rust, but this does not eliminate related programming errors.) By default, integer overflow triggers a run-time panic in debug builds, and *wrap-around* (arithmetic modulo 2^n) in release builds [1]. If this is not as desired, the defaults can be overridden by using built-in integer operation methods: (1) *checked overflow* returns `Some(x)` if a result `x` is representable within the type's range, else `None`; (2) *saturation* returns a result `x` pinned to the maximum or minimum range value closest to the correct result; and (3) *wrap-around overflow* is as per the default, or a variation can furthermore indicate TRUE if wrapping actually occurred (a result tuple is comprised of the value, and a boolean flag).

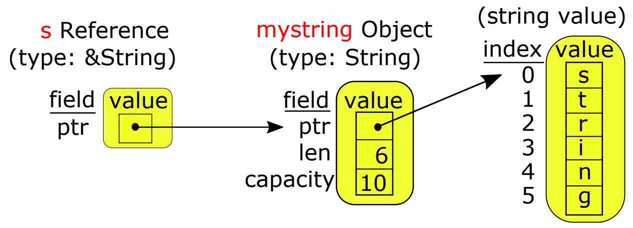

Figure 1: Rust smart pointer for `String` object `mystring` with value "`string`". For function signature `fn_on_str(s: &String)`, the figure shows the result of a call `fn_on_str(&mystring)`. A reference to the object of type `String` is passed, and populates the formal parameter `s`.

The `String` type noted above is an example of a Rust *smart pointer* (as is `Vec<T>`). `String` is implemented as a built-in (C-like) `struct` (see Fig.1). One field is a protected (not subject to alteration, e.g., by pointer arithmetic) raw pointer to a value in heap memory, a second supports bounds-checking (`len`, the size currently in use), and a third helps manage dynamic growth of a string (`capacity`, the maximum size based on memory currently allocated to this object).

Rust `String` objects are well-supported by this combination of pointer (to string value data) plus metadata used to deliver guarantees. In contrast, C strings are programmer-built using a raw pointer to a first `char` and a hope that the following sequence of bytes is terminated by a `NULL` char before the end of the memory allocated for the string.

The `String` type has two so-called *traits* of interest. (Traits provide functionality available for a given type, i.e., functions that may operate on values of that type.) The first is the `Deref` trait, which allows dereferencing the smart pointer (using $\star$).

The second is the `Drop` trait, which specifies the function to call when the object goes *out of scope*, e.g., code to free heap memory. The compiler will generate code to call this function, lightening the programmer's burden (and avoiding programmer errors that enable memory leaks, or release memory twice). The `Drop` trait thus supports implementation of *Scope rule #1*.

Rust's combination of compile time static analysis and automated insertion of code to reclaim heap memory amounts to a *third approach to dynamic memory management*, distinct from both traditional garbage collection, and manual programmer allocation plus deallocation of heap memory.

Other smart pointer types in Rust include the *reference-counting pointer*, the *string slice* (`&str`), and the simplest variety of smart pointer: *box pointer*. This last type, `Box<T>`, allocates a fixed-size object on the run-time stack, holding simply a pointer to a value of type `T`, allocated from heap memory. One common use is when a value is to be moved to a new owner (the new owner simply gets a pointer to the heap value), to avoid inefficient *deep copying* of large data structures.

Memory safety and `unsafe` Rust

Code blocks denoted by the `unsafe` keyword open up a special compiler mode that relaxes some constraints, allowing developers to carry out operations not normally permitted in Rust—including potentially dangerous pointer arithmetic, and dereferencing of raw pointers. The price is thus the loss of some safety guarantees, such as pointers always being non-`NULL` and pointing to valid objects. Rust's hope (and culture) is that programmers strictly scrutinize and minimize use of `unsafe` blocks. This localizes and significantly reduces the volume of code at higher risk of memory vulnerabilities, but the difficulty of finding security flaws by 'scrutinizing' is what motivated using Rust versus C.

Most security issues involving invalid pointers are eliminated by Rust features such as ownership and borrowing rules enforced by the *borrow checker* and static type checking system, smart pointers, and `Drop` traits supporting automated memory deallocation. These allow tracking of object validity and verification that all references (pointers) are valid. In particular, dangling references are flagged as compile time errors. However, some memory errors may still affect Rust code, for example, involving `unsafe` blocks, run-time libraries that may be `unsafe`, and non-Rust components.

Nonetheless, returning to our memory safety levels, spatial and temporal memory errors are largely eliminated (L1). For L2, reading from uninitialized variables is caught as a compile-time error, and concerns about wild (and `NULL`) pointers and invalid heap objects are addressed by Rust's guarantee that all references are validated before use. L3 errors (memory leaks) are also largely eliminated by Rust's approach to dynamic memory management.

Finally, data races (L4) are addressed in safe Rust by its enforcement of *Ownership rule #2*, combined with features supporting multi-threaded programs such as syntax allowing the movement and tracking of object ownership across threads. For further discussion of concurrency and the broader issue of *thread safety* in Rust, see Klabnik and Nichols [9].

While Rust is a big step forward in memory safety for systems languages, security issues remain. One category stems from the reality that while most binaries rely on libraries from a variety of source languages and tool chains, security guarantees are not composable across such components.

Rust delivers guarantees that rely on compile-time analysis, as well as optimizations via eliminating apparently redundant runtime checks—but this combination has been shown to enable use of safe Rust code to exploit non-Rust code in *mixed binaries*, e.g., combining output from C/C++, and Rust compiler tool chains that have differing approaches to delivering memory safety [14].

What remains to be seen is whether Rust will displace C and its cousins in the long term, or ongoing hardening of C via tools and runtime support (software and hardware) will eventually mitigate enough classes of vulnerabilities to weaken the motivation to broadly adopt Rust.

Rust also has a steep learning curve, especially for first-time programmers [6]. On the other hand, there is a big difference between learning enough C syntax to write simple programs, and gaining sufficient experience to avoid writing C code with hidden vulnerabilities. This observation has led to suggestions that while it takes time to become a productive Rust developer, it takes longer to learn C well enough to avoid writing dangerous programs.

References

[1] J. Blandy, J. Orendorff, L. Tindall. *Programming Rust: Fast, Safe Systems Development* (2e). O'Reilly Media, 2021.

[2] L. Cardelli. Type Systems, pp.2208–2236 in: *The Computer Science and Engineering Handbook*, CRC Press, 1997. Revised version in 2nd Edition (2004), `http://lucacardelli.name/papers/typesystems.pdf`

[3] C. Cifuentes, G. Bierman. What is a secure programming language? 3:1–3:15, Summit on Advances in Programming Languages (SNAPL), 2019.

[4] David J. Eck. *Introduction to Programming Using Java*. Version 9.0, May 2022. `https://math.hws.edu/javanotes/`

[5] D. Evans. Using Rust for an undergraduate OS course, 2013. `http://rust-class.org/0/pages/using-rust-for-an-undergraduate-os-course.html`

[6] M. Fluet. Experience report: Two semesters teaching Rust. Pages 45–58 in: Rust-Edu Workshop proceedings, Aug 2022. `https://rust-edu.org/workshop/proceedings.pdf`

[7] 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/`.

[8] P. Holzinger, S. Triller, A. Bartel, E. Bodden. An in-depth study of more than ten years of Java exploitation. 779-790, ACM CCS, 2016.

[9] Steve Klabnik, Carol Nichols. *The Rust Programming Language* (covers Rust 2018). No Starch Press, August 2019. `https://doc.rust-lang.org/book/`

[10] G. McGraw, E.W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. Wiley, 1999.

[11] Matt Miller (Microsoft). Trends, challenges, and strategic shifts in the software vulnerability mitigation langscape. BlueHat Israel Conference, 7 Feb 2019. Slides 1-24. `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`

[12] National Security Agency. Software Memory Safety. Cybersecurity information sheet, Version 1.0, Nov. 2022. `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`

[13] P C van Oorschot. Memory errors and memory safety: C as a case study. *IEEE Security & Privacy* 21(2), Mar-Apr 2023.

[14] M Papaevripides, E Athanasopoulos. Exploiting mixed binaries. *ACM Trans. Priv. Secur.* 24(2) 7:1-7:29, 2021.

[15] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[16] D. Stuttard, M. Pinto. *The Web Application Hacker's Handbook*. Wiley, 2007.

[17] Bill Venners. Design with runtime class information. JavaWorld. Feb 1, 1999. `https://www.infoworld.com/article/2076355/design-with-runtime-class-information.html`