There is no memory safety without thread safety

Memory safety is all the rage these days. But what does the term even mean? That turns out to be harder to nail down than you may think. Typically, people use this term to refer to languages that make sure that there are no use-after-free or out-of-bounds memory accesses in the program. This is then often seen as distinct from other notions of safety such as thread safety, which refers to programs that do not have certain kinds of concurrency bugs. However, in this post I will argue that this distinction isn't all that useful, and that the actual property we want our programs to have is *absence of Undefined Behavior*.

Breaking memory safety with a data race

My main issue with the division of safety into fine-grained classes such as memory safety and thread safety is that there's no meaningful sense in which a thread-unsafe language provides memory safety. To see what I mean by this, consider this program written in Go, which according to Wikipedia is memory-safe:

```
package main

// Just some arbitrary interface so we can later use an interface type.
type Thing interface {
    get() int
}

// Two types implementing the interface, with fields of very different types.
type Int struct {
    val int
}
func (s *Int) get() int {
    return s.val
}

type Ptr struct {
    val *int
}
func (s *Ptr) get() int {
    return *s.val
}

// A global variable of interface type, that we will swap back and
// forth between pointing to an `Int` and to a `Ptr`.
var globalVar Thing = &Int { val: 42 }

// Repeatedly invoke the interface method on the global variable.
func repeat_get() {
    for {
        x := globalVar
        x.get()
    }
}

// Repeatedly change the dynamic type of the global variable.
```

```
func repeat_swap() {
    var myval = 0
    for {
        globalVar = &Ptr { val: &myval }
        globalVar = &Int { val: 42 }
    }
}

func main() {
    go repeat_get()
    repeat_swap()
}
```

If you run this program (e.g. on the Go playground), it will crash very quickly:

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x2a pc=0x468863]
```

Note that the address that caused the segfault is `0x2a`, the hex representation of 42. What is happening here?

This example exploits that Go stores values of interface types like `Thing` as pairs of a pointer to the data and a pointer to the vtable. Every time `repeat_swap` stores a new value in `globalVar`, it just does two separate stores to update those two pointers. In `repeat_get`, there's thus a small chance that when we read `globalVar` *in between* those two stores, we get a mix of a pointer to an `Int` with the vtable for a `Ptr`. When that happens, we will run the `Ptr` version of `get`, which will dereference the `Int`'s `val` field as a pointer – and hence the program accesses address 42, and crashes.

One could construct a similar example using Go's slices, where the data pointer, length, and capacity of the slice are stored in separate words, and reading a half-updated value can lead to an out-of-bounds access.

What about other languages?

At this point you might be wondering, isn't this a problem in many languages? Doesn't Java also allow data races? And yes, Java does allow data races, but the Java developers spent a lot of effort to ensure that even programs with data races remain entirely well-defined. They even developed the first industrially deployed concurrency memory model for this purpose, many years before the C++11 memory model. The result of all of this work is that in a concurrent Java program, you might see unexpected outdated values for certain variables, such as a null pointer where you expected the reference to be properly initialized, but you will *never* be able to actually break the language and dereference an invalid dangling pointer and segfault at address `0x2a`. In that sense, all Java programs are thread-safe.[1]

Generally, there are two options a language can pursue to ensure that concurrency does not break basic invariants:

- Ensure that arbitrary concurrent programs actually behave "reasonably" in some sense. This comes at a significant cost, restricting the language to never assume consistency of multi-word values and limiting which optimizations the compiler can perform. This is the route most languages take, from Java to C#, OCaml, JavaScript, and WebAssembly.
- Have a strong enough type system to fully rule out data races on most accesses,

and pay the cost of having to safely deal with races for only a small subset of memory accesses. This is the approach that Rust first brought into practice, and that Swift is now also adopting with their "strict concurrency".

Go, unfortunately, chose to do neither of these. This means it is, strictly speaking, not a memory safe language: the best the language can promise is that *if* a program has no data races (or more specifically, no data races on problematic values such as interfaces, slices, and maps), then its memory accesses will never go wrong. Now, to be fair, Go comes with out-of-the-box tooling to detect data races, which quickly finds the issue in my example. However, in a real program, that means you have to hope that your test suite covers all the situations your program might encounter in practice, which is *exactly* the sort of issue that a strong type system and static safety guarantees are intended to avoid. It is therefore not surprising that data races are a huge problem in Go, and there is at least anecdotal evidence of actual memory safety violations.

I could accept Go's choice as an engineering trade-off, aimed at keeping the language simpler. However, putting Go into the same bucket as languages that actually *did* go through the effort of solving the problem with data races misrepresents the safety promises of the language. Even experienced Go programmers do not always realize that you can break memory safety without using any unsafe operations or exploiting any compiler or language bugs. Go is a language *designed* for concurrent programming, so people do not expect footguns of this sort. I think that is a problematic blind spot.

The Go memory model documentation is not exactly upfront about this point either: the "Informal Overview" emphasizes that "most races have a limited number of outcomes" and remarks that Go is unlike "C and C++, where the meaning of any program with a race is entirely undefined". You could say that the use of "most" here is foreshadowing, but this section does not list any cases where the number of outcomes is unlimited, so this is easy to miss. They even go so far as to claim that Go is "more like Java or JavaScript", which I think is rather unfair, given the lengths to which those languages went to achieve the thread safety they have. Only a later subsection explicitly admits to the fact that *some* races in Go *do* have entirely undefined behavior (which is very unlike Java or JavaScript).

Conclusion

I would argue that the actual property people care about when talking about memory safety is that *the program cannot break the language*. All these security vulnerabilities caused by memory safety violations are cases where the code did something which isn't even possible in the language specification, such as jumping to some user-provided array and *executing it as assembly code*. The typical term we use for a program that breaks the language like that is *Undefined Behavior*. The moment your program has UB, all bets are off; whether or not an attacker can then control how exactly this UB manifests and exploit it to their advantage is mostly an implementation detail.[2]

In my view, there's a bright line dividing "safe" languages where programs cannot have Undefined Behavior, and "unsafe" languages where they can. There's no meaningful sense in which this can be further subdivided into memory safety, thread

safety, type safety, and whatnot – it doesn't matter *why* your program has UB, what matters is that a program with UB defies the basic abstractions of the language itself, and this is a perfect breeding ground for vulnerabilities.

In practice, of course, safety is not binary, it is a spectrum, and on that spectrum Go is much closer to a typical safe language than to C. It is plausible that UB caused by data races is less useful for attackers than UB caused by direct out-of-bounds or use-after-free accesses. But at the same time I think it is important to understand which safety guarantees a language reliably provides, and where the fuzzy area of trade-offs begins. I am in the business of *proving* safety claims of languages, and for Go, there's not much one could actually prove. I hope this post helps you to better understand some the non-trivial consequences of the choices different languages have made.[3] :)

1. Java programmers will sometimes use the terms "thread safe" and "memory safe" differently than C++ or Rust programmers would. From a Rust perspective, Java programs are memory- and thread-safe by construction. Java programmers take that so much for granted that they use the same term to refer to stronger properties, such as not having "unintended" data races or not having null pointer exceptions. However, such bugs cannot cause segfaults from invalid pointer uses, so these kinds of issues are qualitatively very different from the memory safety violation in my Go example. For the purpose of this blog post, I am using the low-level Rust and C++ meaning of these terms. ↩
2. I am aware that there's a *lot* one can do with this "implementation detail"; that's basically what all the mitigation techniques from basic non-executable stacks to fancy control-flow integrity are doing. But from a principled, formal perspective, those are all just various forms of limiting how UB manifests. We should absolutely keep doing that, but we should also do everything we can to prevent UB from happening in the first place. ↩
3. In case you are wondering why I am focusing on Go so much here… well, I simply do not know of any other language that claims to be memory safe, but where memory safety can be violated with data races. I originally wanted to write this blog post years ago, when Swift was pretty much in the same camp as Go in this regard, but Swift has meanwhile introduced "strict concurrency" and joined Rust in the small club of languages that use fancy type system techniques to deal with concurrency issues. That's awesome! Unfortunately for Go, that means it is the only language left that I can use to make my point here. This post is not meant to bash Go, but it is meant to put a little-known weakness of the language into the spotlight, because I think it is an instructive weakness. ↩

Posted on Ralf's Ramblings on Jul 24, 2025.
Comments? Drop me a mail!