What is memory safety?

I am in the process of putting together a MOOC on software security, which goes live in October. At the moment I'm finishing up material on buffer overflows, format string attacks, and other sorts of vulnerabilities in C. After presenting this material, I plan to step back and say, "What do these errors have in common? They are violations of *memory safety*." Then I'll state the definition of memory safety, say why these vulnerabilities are violations of memory safety, and conversely say why memory safety, e.g., as ensured by languages like Java, prevents them.

No problem, right? Memory safety is a common technical term, so I expected its definition would be easy to find (or derive). But it's much trickier than I thought.

My goal with this post is to work out a definition of memory safety for C that is semantically clean, rules out code that seems intuitively unsafe, but does not rule out code that seems reasonable. The simpler, and more complete, the definition, the better. My final definition is based on the notion of *defined/undefined memory* and the use of *pointers as capabilities*. If you have better ideas, I'd love to know them!

For the purposes of this post, we are generally considering whether a program *execution* is memory safe or not. From this notion, we deem a program to be memory safe if it all of its possible executions are memory safe, and a language to be memory safe if all possible programs in the language are memory safe.

Not in my house

The recent *Systematization of Knowledge (SoK)* paper, the Eternal War in Memory, exemplifies one common way of defining memory safety. It states that a program execution is memory safe so long as a particular list of bad things, called memory access errors, never occur:

1. buffer overflow
2. null pointer dereference
3. use after free
4. use of uninitialized memory
5. illegal free (of an already-freed pointer, or a non-malloced pointer)

The wikipedia page on memory safety provides a similar definition. This is a good list, [1] but as a definition it is not very satisfying. Ideally, the fact that these errors are ruled out by memory safety is a *consequence* of its definition, rather than the *substance* of it. What is the idea that unifies these errors?

No accesses to undefined memory

One unifying idea might be the following: a memory error occurs when the program accesses *undefined memory*, which is memory that the program has not specifically allocated, i.e., as part of the heap (through malloc), stack (as a local variable or function parameter), or static data area (as a global variable). George Necula and his students, as part of their work on CCured, a project aiming to enforce memory safety for C programs, thus proposed that a memory safe program execution is one that never accesses undefined memory. We can assume that memory is conceptually infinitely large, and that memory addresses are never reused. As such, memory that is freed (e.g., by calling **free**, or by popping a stack frame when returning from a function) is never reallocated and stays permanently undefined.

This definition clearly rules out errors 2 and 3, and rules out 4 if we extend our definition of "allocated" to include "initialized." Error 5 is ruled out if you assume that **free** can only be called on pointers to defined memory.

Unfortunately, this definition does not rule out buffer overflows, at least not in practice. For example, assuming a standard stack layout, *Program 1*'s execution would be considered memory safe by this definition:

```
/* Program 1 */
int x;
int buf[4];
buf[5] = 3; /* overwrites x */
```

The definition allows this program because it is writing to legally allocated memory; it is even writing to it at the correct type. The problem is that it is writing to variable **x** by overflowing **buf**, and intuitively this would seem to be memory unsafe.

Infinite spacing

We can slightly extend the definition to rule out the above program by adding, for the purposes of the definition, the assumption that memory regions are allocated infinitely far apart.

Assuming infinite spacing would render the above program to be memory unsafe. For the purposes of our definition we should view **buf** and **x** as allocated infinitely far apart. As such, the **buf[5]** access is to memory outside of **buf**'s region, and into undefined memory, and thus an error. Overflows of objects allocated on the heap, or static data area, are handled similarly. Note that infinite spacing effectively prevents forging pointers directly from integers because it would be infinitely unlikely that you could pick an integer that corresponds to a legal memory region. Emery Berger and Ben Zorn, in their work on the DieHard memory allocator, aim to enforce this definition of memory safety probabilistically, by randomly spacing allocated objects.

While we are much closer to a satisfying definition, we are still not there yet. In particular, the definition would seem to allow the buffer-overflowing *Program 2*, which is a variant of *Program 1*, above.

```
/* Program 2 */
struct foo {
  int buf[4];
  int x;
};
struct foo *pf = malloc(sizeof(struct foo));
pf->buf[5] = 3; /* overwrites pf->x */
```

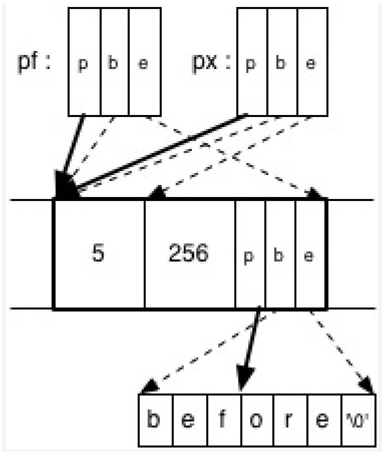
Here, the buffer overflow is *within* an object. We could similarly rule out the buffer overflow by appealing to infinite spacing between record fields. This is not so divorced from reality, as the C standard allows the compiler to decide how to place padding between fields. On the other hand, the language suggests that a record is a single object (a single pointer is returned from **malloc**). Many programs will cast one struct to another, or assume a certain padding scheme. Both operations are supported by many compilers, despite the freedom afforded by the standard. Therefore we would prefer a definition that does not rely on an abstraction that is so from reality.

Pointers as capabilities

The definition I prefer, which I'll now present, is inspired by Santosh Nagarakatte's work (done with several collaborators) on SoftBound, an approach for enforcing memory safety in C programs (which has inspired Intel hardware support, coming soon).

As with our first definition, we have a notion of defined (allocated) and undefined (never-allocated or deallocated) memory, where we assume deallocated memory is never reused, and memory safety is violated if undefined memory is accessed. We add to this a notion of pointers as capabilities; that is, they allow the holder of the pointer to access a certain region of memory. We should think of a pointer as consisting of logically three elements *(p,b,e)*: the legal region is defined by the base *b* and bounds (or *extent*) *e*, and the pointer itself is *p*. The program only truly manipulates *p*, while *b* and *e* are conceptually present for the purposes of defining whether an execution is memory safe.

As an example, here is *Program 3* and a visualization of the (conceptual) memory that results from running it.



```
/* Program 3 */
struct foo {
  int x;
  int y;
  char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```

The last two lines are most interesting. The program is allowed to employ pointer arithmetic to create new pointers, but may only dereference them if they stay within the allotted range *b* to *e*. In the example, we incremented **pc**'s *p* component, but it retains the *b* and *e* components it started with; as such, executing ***(pf->pc)** would be legal. If instead we'd done **pf ->pc += 10**, then ***(pf->pc)** would constitute a violation of memory safety even if **pf->pc** happened to point to defined memory (allocated to some other object).

Program 3's last line creates a pointer **px** to the first field of **pf**, narrowing the bounds to just that field. This rules out overflows like that in *Program 2*. If instead we had retained the bounds of **pf** the program could use **px** to overflow into the other fields of the struct.

A capability is not forgeable, and likewise we should not be able to forge pointers by casting them from integers. Illegal casts could be *direct*, e.g., by doing **p = (int *)5**, [2] or *indirect*, e.g., by casting a struct containing integers to an object containing pointers, e.g., by doing **p = (int **)pf** to effectively cast the first field of the struct in *Program 3*, defined as an integer, to be a pointer. Our definition simply treats casts as no-ops; only legal pointers can be dereferenced, and a pointer's capabilities are determined when it is created. Thus our definition permits *Program 4*.

```
/* Program 4 */
int x;
int *p = &x;
int y = (int)p;
int *q = (int *)y
*q = 5;
```

Conceptually, the base and bounds originally given to **p** will stay with it even when it is converted into the integer **y**, so that they are present when **y** is converted back to **q** and dereferenced. Dually, if we did **p = (int **)pf** at the end of *Program 3* above, we could follow it with ***p = malloc(sizeof(int))** and then be allowed to do both ****p** and **printf("%d\n",pf->x)**. That is,

memory that once contained an integer can be modified to contain a pointer which is then dereferenced, and pointers can be safely treated as integers, but not vice versa.
In a sense, the capability-based definition of memory safety is a form of type safety, where there are only two types: pointer types and non-pointer types. [3] The definition ensures that (1) pointers are only created in a safe way that defines their legal memory region; (2) pointers can only be dereferenced if they point to their allotted memory region; and (3) that region is still defined. And this definition rules out all five error types while allowing for a reasonable set of coding idioms.
What do you think? Does this definition present a good mental model of what one should think of when trying to write a memory-safe C program? Are there important examples of bad code that this definition would consider safe, or examples of the reverse, and if so how should we adjust the definition to accommodate them, while hopefully keeping it simple?

Conclusion

Some of you might be wondering at this point: Has this just been an "academic" exercise? Why is a definition of memory safety useful?

To me, a good definition matters for scientific progress. When I read a technical paper that claims to ensure memory safety, I might think I'm getting one thing when the paper is actually ensuring another. [4] Such confusion impedes progress. For example, when a technical paper develops some technology, e.g., for analyzing C programs, that claims it is correct *assuming* memory safety, we cannot really challenge that claim unless we know what memory safety is. The papers Static Error Detection using Semantic Inconsistency Inference and Large-Scale Analysis of Format String Vulnerabilities in Debian Linux claim to have a *sound* analysis (more on this idea in my previous post on Heartbleed) assuming the underlying program is memory safe. What definition of memory safety are they thinking of?
The thinking contained in this post has benefited from discussions with Emery Berger, Evan Chang, Derek Dreyer, Jeff Foster, Dan Grossman, Milo Martin, Kris Micinski, Andrew Miller, Greg Morrisett, Andrew Myers, Santosh Nagarakatte, James Parker, Aseem Rastogi, John Regehr, Ken Roe, Andrew Ruef, Peter Sewell, Yannis Smaragdakis, Ross Tate, and probably a few others I've forgotten. Thanks, guys!
OK, now back to my course prep ...

Notes:

1. The wikipedia page adds out-of-memory errors to its list of memory safety violations. I don't agree with this addition; type-safe languages like Java, which use garbage collection, are universally viewed as memory safe, and yet they have out-of-memory errors too.
2. For the C/C++ novice: **p = (int*)5** means the pointer **p** is referencing memory address 5, which is (typically) un-allocated memory, and therefore constitutes an illegal pointer.
3. An important proviso is that function pointers may not be the same as normal pointers. I haven't thought long and hard about it, but it would be reasonable to suggest that a program that casts an **int*** pointer to a function pointer and then calls the function is not memory safe. On the other hand, this might actually work: actions taken by those integers interpreted as instructions are fine as long as they follow the rest of the rules. So perhaps **int*-**to-function-pointer is memory safe, but not type safe.
4. I was surprised to find papers like Implementation of the Memory-safe Full ANSI-C Compiler and Memory safety without garbage collection for embedded applications which use the term memory safety in the title, and yet never carefully define the term in their paper. I don't mean to pick on these two papers; they constitute very good work, and are by no means

the only papers not to clearly define what they mean. But they are an example of the problem I am trying to solve here.