Software Security 07

Sven Schäge

Learning Goals

- Be able to reflect on the notions of memory safety and type safety and their relationship
- Identify spatial and temporal memory-safety (in code) and a lack of them.
- Provide examples of programming errors that violate memory-safety.
- Be able to explain and reflect on the mechanisms that make RUST memory-safe and secure against data races at the same time.

Safe Programming Languages

Pointers and Memory Allocation

- Recall that large memory objects that span several consecutive memory units are referenced via pointers.
- A pointer is intialized as the starting address of that sequence of memory units.

Memory-Safety

- Property of a program execution that is memory-safe.
  - 1) Pointers are only created through standard means of the language, e.g. malloc (in C) or new.
  - 2) Pointers are only used to access memory that belong to these pointers. This means that they only access the memory regions that have been allocated for them.
- A program is memory safe if all executions are memory safe.
- A language is memory-safe if all possible programs written in that language are memory-safe.

Memory-Safety

- Spatial memory safety: pointer does not read or write outside of bounds.
- Temporal memory safety: no dangling pointers, i.e. pointers that map to undefined memory regions, no double-free

Type-Safety

- Better than memory-safety.
- Objects have simple or more complex types (e.g. int, pointer to int)
- Operations on object are always compatible with type of object
    - No problems of the following type:
        - Memory may in fact belong to pointer (no memory safety violation!)
        - However, memory is used as wrong type
- A type-safety error essentially means that memory is interpreted differently from what it was intended for.
    - For example, assume we have a variable that holds a 32 integer in some memory location. Now, only reading out the first 8 bits as an integer of that variable is a type safety violation.
    - Such operations can speed up computations considerably.

Variants of Type-Systems

- Most programming languages use type-systems.
- Type systems allow to **commit** to the sets of values (so-called **types**) that variables can hold and how to interpret them.
- If the programmer has to define the types we speak of manifest typing.
- The automatic detection of types is called type inference and we speak of inferred typing.
- Type checking is the process of checking if variables are initialised and exchanged without violating these commitments.
- Type checking can be done at run-time or compile time (static type checking or dynamic type checking).

Variants of Type-Systems

- When more complex types emerge, type systems need to check for compatibility between types.
- If compatible, the type system can implicitly re-interpret/“convert“ types – without the programmer having to do this.
- Complex types not only need to be checked for equivalence but also for subtyping where one type is a subtype of another. This is necessary to support hierarchies of types that are so widespread in object-oriented languages.
- However, in the case of complex types, the type systems usually need additional information on how to compare types.
    - In nominal typing, the type system relies on the explicit definitions of types.
    - In structural typing, the type system relies on the concrete structure of the type to check this. (At compile time).

RUST and Ownership

- RUST avoids memory safety problems since memory resources always have a single owner.
- An owner is a variable/pointer that holds the write access to that memory resource.
- This concept avoids that pointers alias in a bad way
  - This helps to avoid problems in static analysis
- RUST closely keeps track on when ownership changes.
- By definition (literally) it helps to avoid data races (concurrency lecture) as well.