COM-402 exercises 2024, session 5: Language Security

Exercise 5.1

Explain type safety and memory safety.

Exercise 5.2

Is type safety necessary to reduce security vulnerabilities?

Exercise 5.3

Why would a software developer prefer to use a safe programming language? Why would a developer prefer an unsafe programming language?

Exercise 5.4

Does language safety impose a burden on a software developer?

Exercise 5.5

What are the advantages and disadvantages of the memory reclamation techniques discussed in the lecture?

Exercise 5.6

Give examples for violations of spatial and temporal memory safety. Explain how modern programming languages prevent such violations.

Exercise 5.7

Explain static/dynamic typing. What are their advantages and disadvantages?

Exercise 5.8

Compare and contrast dynamic typing and loose (weak) typing.

Exercise 5.9

Do modern programming languages provide full guarantee for thread safety?

Exercise 5.10

Explain the principle of least privilege, sandboxing, and compartmentalization in your own words. Show how they relate to each other.

Solutions to the Exercises

Solution 5.1

Type safety ensures that only type-appropriate operations are applied to values during a program's execution. C-style casts allow arbitrary type casts, relying on the programmer to ensure the cast is valid. In C++, upcasting a class pointer to a base class pointer is safe, but downcasting a base class pointer to a derived class pointer can be unsafe. For instance, if classes B and C both inherit from class A, a pointer to a B object can be safely cast to a pointer to an A object, but casting that pointer to class C is unsafe.

Memory safety ensures that only live memory regions are accessed through a properly acquired (in-bound) pointer. Examples of memory safety violations include out-of-bounds array accesses and use-after-frees.

Solution 5.2

Yes. Type safety helps to prevent certain software defects that could be exploited in attacks. The lecture slides demonstrate that a type confusion bug can be exploited in the Chrome browser. Additionally, memory safety requires type safety to ensure an object's class type can be checked, confirming that a struct or object field reference is legal.

Solution 5.3

A developer should prefer a safe programming language if they are developing software with a publicly exposed attack surface because language safety can prevent many types of software defects, such as buffer overruns (Google's "rule of 2"). Developers might also prefer safe languages as they can aid in writing robust and reliable software by exposing software defects.

A developer might prefer an unsafe programming language because they believe it allows writing more efficient code, optimizing a computer's processor and memory usage.

Solution 5.4

Yes, but it is a necessary burden. Safety requires developers to ensure that their programs' types and pointer references conform to a language's rules. These rules ensure programs behave predictably and according to language semantics without corrupting data or performing ill-defined operations. Safe languages facilitate this by helping developers identify defects with compiler and runtime error messages. While it's possible to write correct programs in unsafe languages, it's more challenging without compiler and runtime assistance. There are cases where a program with type or memory errors might produce the "right" answer, but such errors can cause the program to misbehave with other inputs.

Solution 5.5

Manual memory allocation and deallocation (e.g., malloc/free in C) grant a developer complete control over object lifetimes, potentially optimizing a program's memory usage. However, accurately tracking object lifetimes is complex and prone to errors, which can introduce defects.

Reference counting effectively reclaims non-cyclic data structures. With efficiency improvements through language and compiler assistance, it reclaims objects when they first become unreachable, without long pauses. However, explicit cycle breaking is required for full memory reclamation.

Garbage collection has potentially lower performance overhead than reference counting and can reclaim cyclic data structures. It might also improve program performance by compacting data structures to fit the processor cache better. However, garbage collection usually introduces pauses in program execution to scan memory, which might be unsuitable for certain applications.

Ownership and borrowing (as in Rust) tie object lifetimes to variable scopes. Like smart pointers in C++, they release an object when its encompassing variable goes out of scope. While ownership incurs a low cost, it might not meet all allocation and deallocation requirements, even with borrowing. Rust also offers reference counting for data structures whose lifetimes aren't tied to a specific variable.

Solution 5.6

Spatial memory safety violations involve accessing memory locations that a program shouldn't, such as out-of-bounds array accesses. Temporal memory safety violations occur when a program accesses memory at a time it shouldn't, like accessing memory after it has been freed (use-after-free), double freeing memory, or returning a pointer to a local (stack) variable.

Modern programming languages employ various techniques to prevent these violations. For spatial memory safety, many languages enforce bounds checking to ensure data is accessed within its allocated space. For temporal memory safety, languages may use techniques like garbage collection (as seen in Java) or ownership and borrowing systems (as in Rust) to manage memory access and deallocation.

Solution 5.7

Static typing checks types at compile time, whereas dynamic typing checks types at runtime.

Advantages of static typing: (1) Can catch type errors early, often leading to more robust software. (2) Can lead to performance optimizations since type information is available at compile-time. (3) Provides clearer documentation via type annotations, helping developers understand code better.

Disadvantages of static typing: (1) Can be more verbose as type annotations are usually required. (Type inference can help reduce verbosity) (2) Might require additional boilerplate, e.g., using templates in C++ for generic programming.

Advantages of dynamic typing: (1) Provides flexibility and allows for rapid prototyping. (2) Reduces verbosity as type annotations are often optional.

Disadvantages of dynamic typing: (1) Typing errors can only be detected at runtime, which can be problematic for large and complex programs. (2) Absence of type annotations can make code harder to understand and maintain.

Solution 5.8

Dynamic typing pertains to when type checking occurs: at runtime. Loose (or weak) typing, on the other hand, describes how strictly type rules are enforced allowing for more implicit type conversions.

It's essential to understand that dynamic typing and weak typing are orthogonal concepts. A language can be dynamically typed and either strictly or loosely typed. For instance, Python is both dynamically and strictly typed because, while type checking happens at runtime, it enforces type rules rigorously. Conversely, C is statically and weakly typed, as it allows arbitrary pointer casts, implicit type conversions and performs type checks at compile-time.

Solution 5.9

No, modern programming languages do not provide a "full guarantee" for thread safety. Thread safety is about ensuring multiple threads can access shared resources without causing unintended behaviors or data inconsistencies. Although many modern languages offer features and constructs (like 'synchronized' in Java, goroutines and channels in Go, and ownership in Rust) to aid in achieving thread safety, these are tools and not guarantees. For example, while modern programming languages offer locks in their standard libraries, there's no assurance that programs using these languages will be free of deadlocks.

Solution 5.10

The principle of least privilege dictates that a user, program, or process should possess only the minimum privileges needed for its function.

Sandboxing is a security technique wherein a program or process operates in a restricted environment with limited resource and system function access.

Compartmentalization involves dividing a system into isolated sections, ensuring a breach in one doesn't compromise another.

Both sandboxing and compartmentalization help enforce the principle of least privilege by restricting program or process privileges to only those necessary. In a programming language context, compartmentalization is often more fine-grained, operating at function or library levels, whereas sandboxing is more coarse-grained, limiting access at the process level.