SWE 4503

Memory and Type Safety

Definitions, Significance

Memory safety ensures that pointers always point to a valid memory object, i.e., **each memory access is in bounds.**

Type safety refers to ensuring that a program only manipulates data in ways that are consistent with its data types. **Simply, you can not treat an integer like a float.** In strongly-typed languages like Java, Rust, or Haskell, the compiler ensures type safety by enforcing strict rules on how data can be used. Here's an example illustrating both type safety and a violation of it.

Bugs that cause the program to **violate memory or type safety** can be used to **change the runtime state of the program**. This **modified runtime state** leads to an **execution that would not be possible under a benign execution** of the program. For example, **instead of encoding a .jpg image into a .png image, an image web application may upload personal credentials like user id and passwords.**

Difference between High and Low Level languages

High level language guarantee memory and type safety and thus limit attackers to a given specification and constraints of the implementation. It means bugs are constrained and can only violate logical flaws.

Memory unsafe (low level) languages like C do not enforce memory or type safety and data accesses can occur through stale/illegal pointers.

Pointers Capabilities

Before going into safety violations and fix we need to know about pointers.

- **Pointers** are **unstructured addresses to memory** and a way to reference data or code.
- A **pointer** has an **associated type and a value, the address** it points to.
- Under C/C++ pointer arithmetic allows modification of a pointer through increments and decrements.
  - The validity of the pointer is not enforced through the programming language but must be checked by the programmer. For example, **after a pointer increment the programmer must ensure that the pointer still points into a valid data or code snippet.**
- The **capabilities** of a memory object describe the **size** or area, **validity**, and potentially the **type** of the underlying object. Capabilities are assigned to a memory object when it is created. Like, **int v[10], here v is a memory object where integer is the type, 10 is the size of v and validity is until the program terminates.**
- **Pointer capabilities** cover three areas: bounds **(spatial safety)**, validity **(temporal safety)**, and type **(type safety)**. **All these are for pointers.**

Continued

- **Spatial safety** ensures that pointer dereferences are restricted to data inside of the memory object.
- **Temporal safety** ensures that a pointer can only be dereferenced as long as the underlying object stays allocated.
- **Type safety** ensures that the object's type and pointer type matches.

The C/C++ family of programming languages allows invalid pointers to exist, i.e., a pointer may point to an invalid memory region that is out of bounds or no longer valid. A memory safety violation only occurs when such an invalid pointer is dereferenced. We already know about pointers but let's try function pointers.

Function Pointers

```c
#include <stdio.h>

void neverCall() {
    printf("You are DOOMED");
}

void call() {
    printf("You are fine");
}

void (*ptr)() = &neverCall;

int main() {
    ptr();
}
```

```c
#include <bits/stdc++.h>
using namespace std;

class TypeSafety{

public:

    TypeSafety(){}
    void neverCall(){
        printf("You are DOOMED");
    }

    void call(){
        printf("You are fine");
    }

    void (*ptr1)() = &neverCall;

    void (*ptr2) () = &call;
    ~TypeSafety(){}
};

int main()
{
    TypeSafety obj;

    obj.ptr1();

    return 0;
}
```

Memory Safety Prohibits

- Buffer Overflows
- Null Pointer Dereferences
- Use after free
- Use of Uninitialized Memory : can lead to undefined behavior, which means the program could crash, produce incorrect results, or behave unpredictably.
- Double frees : when the free() function is called more than once on the same memory address. This is dangerous and can lead to undefined behavior, such as program crashes, heap corruption, or security vulnerabilities.

Language based memory safety makes it impossible for the programmer to violate memory safety by, e.g., checking each memory access and type cast (Java, C#, or Python).

Buffer Overflows

```c
#include <stdio.h>
#include <string.h>

void unsafe_function(char *input) {
    char buffer[8];  // Small buffer

    // Unsafe: doesn't check if the input fits the buffer
    strcpy(buffer, input); // Potential buffer overflow
    printf("Buffer content: %s\n", buffer);
}

int main() {
    char large_input[20] = "This is too long";
    unsafe_function(large_input);  // Causes buffer overflow
    return 0;
}
```

Buffer Overflows Fix

```c
#include <stdio.h>
#include <string.h>

void unsafe_function(char *input) {
    char buffer[8];  // Small buffer

    // Unsafe: doesn't check if the input fits the buffer
    strcpy(buffer, input); // Potential buffer overflow
    printf("Buffer content: %s\n", buffer);
}

void safe_function(char *input) {
    char buffer[8];

    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';  // Ensure null-termination
    printf("Buffer content: %s\n", buffer);
}

int main() {
    char large_input[20] = "This is too long";
    //unsafe_function(large_input);  // Causes buffer overflow
    safe_function(large_input);
    return 0;
}
```

Null Deferences

A null pointer dereference occurs when a program tries to access or modify data through a pointer that is null (i.e., pointing to nothing).

```
int *nullDeferencing;
*nullDeferencing = 10;
```

Present compilers both C/C++ are well protected from null dereferencing in most cases like the above.

Use After Free

After free() the pointer becomes a dangling pointer that means it still holds the address of the now deallocated memory, which is dangerous.

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int *ptr = (int*) malloc(sizeof(int));
    free(ptr);
    *ptr = 10;

    printf("Integer value %d\n",*ptr);
}
```

Use After Free Fix

```
#include<bits/stdc++.h>
using namespace std;


int main(){

    int *useAfterFreePtr = (int*) malloc(sizeof(int));
    free(useAfterFreePtr);
    useAfterFreePtr = NULL;
    *useAfterFreePtr = 10;

    printf("Use After Free Pointer is %d\n",*useAfterFreePtr);

    return 0;
}
```

Spatial Safety Violations

These violations happen if a pointer is

- Incremented past the bound of the object
  - In a loop
  - Through pointer arithmetic
- Direct Write
- Deferenced

The C/C++ standard allows pointers to become invalid as long as they are not used. A pointer can be incremented past the bounds of the object. If it is decremented, it may become valid again. **Note that a pointer may only become valid again for spatial safety. If the underlying object has been freed, the pointer cannot become valid again.**

```
#include<bits/stdc++.h>
using namespace std;

int main(){

    char *c = (char*) malloc(24);
    for(int i=0;i<24;i++)c[i] = 'A'+i;

    c[26] = 'A';
    c[-2] = 'Z';

    char *d = c+26;
    d -= 3;
    *d = 'C';

    printf("Character is %c\n",*d);

    return 0;
}
```

Temporal Safety

Various forms of temporal safety violations exist. One of them is, a stale pointer can be used after the underlying object has been returned to the memory allocator and even after that memory has been reused for a different object.

```
#include<bits/stdc++.h>
using namespace std;

int main(){

    char *c = (char *)malloc(26);
    char *d = c;
    free(d);

    c[23] = 'A';

    printf("Character is %c\n",c[23]);

    return 0;
}
```

How Java maintains Spatial and Temporal Safety

Spatial Safety

- Java throws ArrayIndexOutOfBoundsException if any array element out of its declared range is accessed.
- Java throws NullPointerException if we attempt to dereference a null reference.

Temporal Safety

- Java's garbage collector eliminates the risk of dangling pointers (accessing memory after it has been freed). Once an object is no longer referenced, the garbage collector eventually reclaims the memory.
- Java does not allow the programmer to free memory manually (as C/C++ does with free() or delete), there is no risk of double free or freeing memory that has already been freed.

SoftBound and **CETS** enforces spatial and temporal safety for C/C++ with different trade-offs regarding performance, memory overhead and hardware extensions.

Type Safety

- **Well-typed programs cannot “go wrong”.** (Robin Milner, Turing Award 1991, made significant contributions to areas related to programming languages)
- Full type safety does not imply memory safety. The two properties are distinct.
- Upward casts (upcasts) move the type closer to the root object, the type becomes more generic, while downward casts (downcasts) specialize the object to a subtype.
  - Upward : int* -> void*
  - Downward : void* -> int*
- For C, the type lattice is fully connected, any pointer type can be cast to any other pointer types with the validity of the cast being the responsibility of the programmer.
- In C++ there are several casting operations. The most common is static_cast.

Type Safety Violations & fix

```c
#include <stdio.h>

void show_number_float(float* fptr) {

    printf("Reinterpreted value: %f\n", *fptr);

}

void show_number_int(int* iptr) {
    printf("Reinterpreted value: %d\n", *iptr);
}

int main() {

    int number = 10;
    void* ptr = &number;

    show_number_float(ptr);
    show_number_int(ptr);

    return 0;
}
```

```c
#include <bits/stdc++.h>
using namespace std;

class TypeSafety{

public:

    TypeSafety(){}

    show_number(int* iptr){

        printf("Showing Integer : %d\n",*iptr);

    }

    show_number(float* fptr){

        printf("Showing float : %f\n",*fptr);

    }

    ~TypeSafety(){}
};

int main()
{
    int num = 10;

    void* ptr = &num;

    TypeSafety obj;

    obj.show_number(ptr);

    return 0;
}
```

Type Safety Violations

```
#include<bits/stdc++.h>
using namespace std;

class Conv {bool friend;};

class Bye: public Conv {
public :
    void speak(){
        cout<<"Bye!!!"<<endl;
    }
};

class Hi: public Conv {
public :
    void speak(){
        cout<<"Hi!!!"<<endl;
    }
};
```

```
int main(){

    Conv *c1 = new Hi();
    Conv *c2 = new Bye();
    Hi *h;

    h = static_cast<Hi*>(c1);

    if(h) h->speak();
    else cout<<"Casting Error 1"<<endl;

    h = static_cast<Hi*>(c2);

    if(h) h->speak();
    else cout<<"Casting Error 2"<<endl;

    delete c1;
    delete c2;

    return 0;
}
```

What is the Fix then?

In low-level languages like C or C++, type safety is not explicit and a memory object can be reinterpreted in arbitrary ways.

Static casts lack any runtime guarantees and objects of the wrong type may be used at runtime. As shown in the previous slide.

The fix is dynamic_cast which performs a runtime check to ensure that the cast is valid and will return nullptr for pointers or throw an exception for references if the cast is invalid. The trade-offs of this technique is that it requires polymorphism (those that have at least one virtual function) and incurs overhead due to runtime checking.

Type Safety Fix

```cpp
#include<bits/stdc++.h>
using namespace std;

class Conv {
    bool friend;
public:
    virtual void speak(){
        cout<<"Not Saying"<<endl;
    }
};

class Bye: public Conv {
public :
    void speak(){
        cout<<"Bye!!!"<<endl;
    }
};

class Hi: public Conv {
public :
    void speak(){
        cout<<"Hi!!!"<<endl;
    }
};
```

```cpp
int main(){

    Conv *c1 = new Hi();
    Conv *c2 = new Bye();
    Hi *h;

    h = dynamic_cast<Hi*>(c1);

    if(h) h->speak();
    else cout<<"Casting Error 1"<<endl;

    h = dynamic_cast<Hi*>(c2);

    if(h) h->speak();
    else cout<<"Casting Error 2"<<endl;

    delete c1;
    delete c2;
    return 0;
}
```