Software Assurance Tips

A product of the Software Assurance Tips Team[1]

Jon Hood

Monday 10th April, 2023

1 Ada Unchecked Conversions

Updated Friday 12th May, 2023

The SwA team lives in a privileged area. If you throw a rock out of your window, chances are that you'll hit a C, C++, or C# developer. But it takes a tactical nuke to pinpoint a good Ada developer here. Because of this, several companies have projects that are built partially in Ada and partially in C++. Interactions between these components often cause security issues, but the most misunderstood of them all is the `unchecked_conversion`.

When evaluating software, we often flag the use of `unchecked_conversion` for multiple reasons:

- Sending dynamic memory outside of the scope or locality of where it is instantiated is a violation of RAII programming.
- You must be able to know the exact structure all endpoints that access the memory object are expecting.

1.1 A Violation of RAII

The first issue has to do with RAII principles. When memory objects are created in one locality of the code and used in another, the area of code that created the memory object is no longer in charge of making sure that object is cleaned up. If it does clean up the memory before exiting, the unmanaged locality of code may still be trying to use it! This would cause issues with race conditions, accessing memory that has already been freed, or even accessing new memory objects that are now in that location of memory (and potentially should not be accessible by the other locality of code, ex: the Dirty Cow vulnerability for Linux).

Failing to enforce memory cleanup at the unmanaged endpoint would also result in memory leaks. This could cause crashes and degraded performance over time.

1.2 A Violation of Portability

Code that does not behave the same across platforms violates the concept of portability. This is often the case when `unchecked` conversions and pointers are used. When performing an unchecked data access on memory, the developer must know:

- the endianness of the unmanaged portions of the code
- the compiler options used
- that updates to the unmanaged portion of code will not violate the contract of how the memory should be stored

Consider an Ada program sending a simple data structure to a C program consisting of a character and an integer as defined in Listing 1. The character takes up 1 byte and the integer takes up 4 bytes. So, in the Ada code, we convert the first byte to the character and the next 4 bytes as the integer (accessing both with the `unchecked_conversion` capability. But is that how it really works?

```
struct A
{
        char a;
        int b;
} tmpA = {'a', 1};
```

Listing 1: Simple Structure Example

Of course not! By default, most compilers (like GCC) pad each data type to the next memory alignment. The character takes up 1 byte, then 3 bytes of padding are added before the 4 bytes that define the integer. If you were to run `sizeof(tmpA);` in any modern GCC compiler, the result would be 8 bytes. Or at least, that's the default case. If you were to add the `__attribute__((packed))` GCC directive to the struct, use `#pragma pack`, or compile with `-fpack-struct`, the `sizeof(tmpA)` becomes 5 bytes, misaligning the integer value with memory so that it takes two memory operations to read the integer but taking up less memory.

1.3 Fixes

Solutions that do not use `unchecked` memory access are preferred. This often requires the code to be rearchitected to abide by RAII programming concepts. In lieu of fixing the code, a contract can be documented that guarantees that all parties involved will abide by a particular memory sharing structure.

1.4 Conclusion

When performing unchecked conversions, using external memory access, or violating RAII with two distinct functional objects, a contract should be established. Part of that contract is a guarantee about the memory structure of what is being shared. That contract should include architecture information, memory packing, and even the flags and versions of compilers used to build the memory structures. Creators of this documentation should ask themselves, "What prevents someone from compiling this code with options like `-fpack-struct`?" "How are memory structures updated in newer versions of the software?" and "What is the enforcement mechanism to ensure that the code is only built in the approved way?"

References

[1] Jon Hood, ed. SwATips. https://www.SwATips.com/.