ISO-IEC JTC 1/SC 22/OWGV N0048r

Contribution from Steve Michell, "Vulnerability Issues from TR 15942"
11 December 2006
It had the filename "issue-list-tr-15942.pdf"
It was accompanied by a source file "issue-list-tr-15942.doc"

This revision replaces the previous document which was in txt format and dated 26 October 2006.

Vulnerabilities Issues from TR15942

EXPLANATION

The following Issues are largely collected from International Technical Report 15942:2000 Guidance on the use of the Ada Programming Language in High Integrity Systems. The issues themselves came from Ada and have been couched largely in terms of Ada, but I have attempted to broaden the scope to explain the general issue and to make some statements about how they might manifest themselves in other programming languages.

ISSUES LIST

Section 5.1 Types with Static Attributes

SM 001 Strong typing vs weak typing

from Sect 3.2, 5.1

The choice of storage used to support an algorithm is a trade-off between the ~~possible~~ underlying representations possible on the machine, the efficiency of access associated with those underlying representations, and the language/compiler/tool support available to support the choices made. Most languages choose a tradeoff that~~which~~ maps one of a few fixed-size representations for integer-based types, real numbers, characters, booleans and other types.

On the other hand, the algorithm required usually has well-known properties for range, boundedness, and precision, often many representations~~.~~

All digital programming language systsms make compromises which can result in vulnerabilities.

If the usual range of the algorithm fits within a chosen representation size but exceptional processing may exceed that size, there is a risk that exceeding the size may cause truncation of results (usually known as wrap-around), the generation of an exception, or unexpected change of representation to a larger size. For HI systems, it is usually undesirable to dynamically determine if such situations can occur, so static analysis and choice of representation are used to ensure that this does not occur. (Note that such overflows could also occur during expression evaluation on a partial result even if the final result can be shown to be within bounds).

If the usual range of the algorithm fits always within the chosen storage, there is still a risk that some results will exceed the algorithm bounds and cause chaotic behaviour. Therefore HI systems should be able to state and determine the relative bounds of types used in calculations and ensure that these bounds are not exceeded, except possibly during expression evaluation before a final result is determined. For languages with strong type-checking, good algorithm design can support static determination of most (if not all) calculations as long as the correctness of the inputs to those calculations can be guaranteed. For languages with weak type checking, auxillary tools and additional annotations can be used for static analysis of the algorithms, and explicit runtime checks can be used to support the dynamic verification of the algorithms.

Usually the bounds and operations of one type have no relation to those of another type, unless they are com~~n~~bined in controlled ways. Some characteristics are obvious, such as never performing boolean operations on integers or integer operations on booleans. Others are less obvious such as adding centimeters and inches. Language systems that support the separation of such concepts will not require additional tooling or annotations to show the correctness of the implementation of the algorithm.~~: ~~Llanguage systems without strong typing will require external tools and extended analysis to verify the correct usage of objects.

When static checks are insufficient and runtime checking is required, weakly typed languages or strongly typed languages with runtime checking disabled will require visible (i.e. application) checks of legal values to ensure correct operation of the algorithm.

For many algorithms, the range used by the representation chosen does not use the complete storage of the memory used. The excess memory is never used by the algorithm, and could be available to deliberate or accidental use to carry information. There is not much risk in ranged types since such information would affect range tests, but is possible for simple non-mathematical types or for composite types. This risk is non-existent for strongly typed languages since the unused portion is not addressable from within the algorith and conversion between this type and types which could access these portions is illegal. For weakly typed languages, additional tooling or explicit checks that unused portions are always a known value (say null) would be required to prevent such a vulnerability.

SM 00 2 Unbounded types

Sect 5.1

All objects are bounded. Simple objects such as integer types have word size or multi word sizes and rules about conversions between.

Facet : Static Analysis

SM 003 Runtime support for typing

Sect 5.1

When support for the typing mechanism requires runtime artifacts, requires additional processing and reduced efficiency, makes static analysis less predictable.

SM 004 Arrays

Sect 5.1

Arrays consist of a set of storage for replicated data together with possibly a set of bounds for each dimension. The major issues for language systems for arrays are as follows:

Static or dynamic bounds - In strongly typed systems, static bounds and invisibility of the explict storage make arrays secure.

For strongly typed systems with dynamic bounds, the bounds are not directly accessible but attempts to exceed the bounds will result in exceptional processing.

In weakly typed systems, arrays which should be statically bounded can often be cast to other forms of access, and access outside the bounds is also possible. Tooling or explicit runtime checks are required to ensure that this does not occur.

In weakly typed systems, arrays which can be dynamically bounded will require explicit bounds to be maintained. These bounds can be changed by the application, resulting in inappropriate access to memory.

For some language systems, the access to the storage region containing the object can be manipulated in ways other than access through the base object and an index. For HI systems, Tools and static analysis is required to show that this does

not happen.

SM 005 Objects with variant structure

Sect 5.1

Most programming languages have ways ~~of~~to permit~~ting~~ a contiguous set of storage locations to be viewed in different ways at different times within the application. The most common application-visible way to accomplish this is the union (C/C++) or variant record (Ada, Pascal), though both also have a way to also do “unions”~~.~~.

In weakly typed systems, or in unconstrained objects in strongly typed systems, the view of the object can be arbitrarily changed by the application, which may permit values in one view to be viewed or changed in a different view, and there may be gaps or portions of the object in one view which are not overwritten by writes to a different view.

Also, the size of such an object in one view may differ from other views, permitting possible hiding of data in an otherwise legal application.

In HI systems, it is recommended that multiple views of the same object be forbidden.

SM 006 Name overloading, operator overloading, overriding

Sect 5.1, 5.3

Overloaded names help to~~s~~ preserve human cognitive space, if all items with the same name perform the same basic algorithm. Statically determinable overl~~a~~oaded names can be successfully evaluated by tools, but humans trying to evaluate calls to such overloaded subprograms ~~(especially operators)~~ may experience difficulty determining the correct call from all calls possible, and while tools can determine which of N calls is being made, they cannot tell if it was the intended call by the human. Similar issues exist in languages that have a single name space but case sensitive names, as two names with the same spelling but different casing could be mistakened by humans.

In HI systems, it is preferrable to give unique names to entities or to use tools and likely annotations to show static~~al~~ly that the entities have the same behaviour.

SM 007 Unbounded objects

Sect 5.1

Some languages can produce objects that have sizes which are nonstatic or even unbounded. This discussion does not include objects which are bounded but the language does not check bounds on every access.

Unbounded objects include objects with no embedded bounding mechanism and those with embedded bounding mechanisms. In either case, dynamic memory techniques are required to allocate the object and deallocation after a copy of an object may leave a valid reference to deallocated space.

Facet : Dynamic storage techniques

Section 5.2 Declarations

SM 008 Constants

Constants take the following forms in Ada:

Any object declared a constant in the declarative part of a package or subprogram

Any "in" parameter of a subprogram (either explicitly declared "in" in a procedure or all parameters a of a subprogram.

Any "in" formal parameter of a generic

Any loop iteration variable

Constants are intialized at the point of declaration.

Language rules prohibit the explicit assignment to constants, except as part of the constant declaration/creation process.

General statements about other language forms of constants needed

SM 009 Uninitialized variables

The declaration and initialization of a variable can either occur in a single place or as two distinct steps. Issues for the initialization of objects:

An object with an unknown value before its first use in an expression represents a serious vulnerability, with possible unbounded behaviour resulting from access to such objects before initialization.

Initial values of variables should never be left to chance. Many systems "zero" global memory as the program is beginning, but applications must not rely on this since:

zero may not be a legal value ~~and since~~

any environmental change could result in non-zero values for variables, and

~~-~~objects declared on a stack or in other non-global areas are unlikely to be initialized.

Where the object can be initialized as part of a declaration, this should be done. In languages such as Ada, there is a phase before subprogram execution commences (such as in elaboration phase or package body execution) where this elaboration can be done. In languages without this intermediate place, applications must determine where the first access in the complete program will occur and ensure that initialization occurs prior to that event (this may be a challenging computation).

Some applications~~systems~~ prefer initial illegal values be declared to support testing, but careful thought should be given to this approach as leaving this approach in operational systems could cause unplanned exceptional behaviour, or cause a substantial change between tested code and operational code.

SM 010 Aliasing

Aliasing of a variable (access via multiple paths) makes it difficult to verify that the variable is being updated or accessed correctly. Aliasing can result from

access to objects through access types (pointers),

having local (via a parameter) and global view of an object, and

making the same object an actual parameter for multiple parameters in a single call.

Ada's copy-in/copy-out semantics for subprogram and entry parameters eliminates some problems associated with order of access, and the ability to construct and use compound objects as such parameters eliminates many access types in Ada. Applications must still show, however, that aliasing does not occur, or that it is correctly identified and handled if it does occur.

Languages with “reference” approach to exportable parameters must show through static analysis that the actual object (i.e. that passed as the parameter) is not also accessed from within the subprogram. In general this is a hard computation, and may require formal annotations.

Section 5.3 Names

SM 011 Nested subprograms

Some languages permit subprograms to be textually nested inside other subprograms. Such nesting makes test coverage almost impossible except in simple cases. Nested subprograms also have the property that local variables of one subprogram are visible from nested subprograms and may be accessed directly instead of being parameterized.

Section 5.4 Expressions

SM 012 Expressions on objects of composite types

Some languages permit operations on objects which cause significant nonvisible code to create, copy, compare. This could cause problems in timing analysis or in object code analysis.

On the other hand, operations on composites where the language does not support whole-object operations mean that each component of the object must be explicitly created, meaning that static an~~-~~alysis must be performed to show full coverage. This presents special challenges during maintenance when new components can be added.

SM 013 Expressions on multiple conditions.

Potential problems with order of evaluation, unintended casting, short-circuit forms

SM 014 Object slices

A slice of an object is a contiguous part of it. When the target and the result of an operation target parts of the same object and those parts overlap, competing access to the same location may create errors. Such access will likely be problematic for static analysis tools.

Where slices are defined in a language, dynamic bounds for slices are problematic for static analysis tools.

SM 015 Goto Statement

Static analysis of code almost always assumes standard control constructs. Use of Goto when using these tools causes code to be intractable for these kinds of analysis.

The usual place that goto is used in some languages is to escape from deeply nested control structures where an alternative construct is absent. In those languages, such as C, C++, Java(?), the alternative constructs may be more confusing then the use of the breakout goto. In languages with good alternative constructs, such as Ada which has "exit Loopname", there should be no need for "goto".

Section 5.5 Statements

SM 016 Loop statements

Loop statements include

the loop controls mechanism

the loop start

loop end .

Simple loops with static control mechanisms and well-defined start and end mechanisms have almost no issues with any analysis mechanisms or cognitive issues.

SM 017

For loops with static bounds, and where analysis can show that no modification of the loop control variable is possible are similarly analysable and safe. For a language such as Ada, language rules guarantee most loop properties, except that dynamic ranges for the loop control variable could make timing more difficult.

SM 018

For languages where the control variable step function may be an arbitrary expression, static analysis of the loop control expression may be intractable.

SM 019

For languages where the control variable termination function may be an arbitrary function or may be dynamic, static analysis of the loop control expression may be intractable, and combined with d), arbitrary loop increments and arbitrary termination expressions may cause non-terminating loops.

SM 020

For languages where assignment to the control variable is permitted, static analysis of the loop control expressions may be intractable.

Recommend that HI systems only permit static expressions for loop start, increment and termination

SM 021

Loops with embedded exit conditions usually protect the exit with some kind of conditional test. The placement of such an exit (inc goto) and the nature of the test may make timing analysis difficult.

5.6 Subprograms

SM 022 Function side-effects

Functions which have only "in" variables and which update only local variables are side-effect free, safe, and amendable to static analysis. Functions with parameters that are access types or explicit "var" (pass-by-reference) parameters provide a vehicle for the program to update alaised objects through those parameters, and updates to non-local objects destroy the side-effect-free aspect of functions.

HI Applications should always document all input and output to all subprograms. For those subprograms where the access or update is through accss parameters or through non-local objects, this must be documented through comments or non-programming mechanisms.

Order of Evaluation - A predictable order of evaluation is fundamental for showing correct behaviour of high integrity systems. We identify the following order of evaluation classifications and their issues:

SM 023 Expression order of evaluation

This subdivides into issues of precedence, which are handled in ???, and evaluation order where precedence is not an issue.

Where the language specifies evaluation order in all cases, the application can depend upon that order; for those languages or situations where the order is not specified, applications must be written such that order of evaluation does not matter. In fact, it is recommended that expressions always be written such that the order of evaluation of expressions does not affect the correctness of the algorithm.

Explicit use of brackets to control evaluation order for complex expressions should be considered carefully. Too many levels of brackets cause as much confurion for the human reader as do too may expression terms. Reducing expression complexity by dividing them it multiple statements is often superior to heavy use of brackets.

SM 024   Parameter order of evaluation

Where actual parameters of a subprogram contain expressions, if subprograms can have side effects, or for possibly aliased components, the order of evaluation of those parameters can effect the correctness of the execution of the subprogram. For languages with copy-in/copy-out semantics and specify parameter order for subprograms, avoiding access types (pointers) in actuals, access parameters, and actual parameters which name the same object effectively eliminates evaluation order issues. For languages with pointer semantics for "out" parameters as well as cases where the actual parameter is an access type, applications must be written such that order of evaluation upon subprogram call or return is irrelevant to the correct operation of the subprogram.

SM 025 Subprogram parameters - Aliasing

Some use local-copy/aliased actual-model, some use local-copy/copy-in-copy-out/aliased-actual model. Use of aliased actual means that update of actual occurs immediately when the parameter is updated, and may leave actuals of subprogram inconsistent if exception or context switch occurs.

SM 026 Subprogram parameter matching

Ada's subprogram parameters are intimate with the strong typing of the language: each call to a subprogram statically matches the type of each parameter with the specification of the subprogram, and the implemation must also statically match. In addition, Ada's named parameter calling convention helps eliminate mistakes when similar or overlapping types may be used in the same call, or when the order or number of parameters in a subprogram may change during maintenance.

For languages which are more~~less~~ permissive, tools must be used to guarantee that every subprogram call statically matches the specification of the subprogram, and that the implemation of the subprogram matches the specification (this includes verification of the type of each parameter, possibly the range of each parameter and the number of parameters). Where positional notation is the only way of creating a subprogram call and the types of the actuals of the call overlap, additional annotations may be useful (necessary?) to help static checking tools verify that the code matches the intent.

Section 5.7 Packages

SM 027 Aliasing of subprogram parameters

Special case of above issue, but aliasing of some object by 2 or more parameters is problematic.

~~(aside – missing some issues about name space, information hiding)~~

(aside – missing some issues about name space, information hiding)

5.8 Arithmetic Types

SM 028 Integer Types

There are a number of issues for integer types. The only issues arising from Ada's Integer types occur in evaluating expressions that can result in the expected range being exceeded. In other languages, other issues must be addressed, such as silently exceeding the safe range of an object (usually tied to a word size) causing wraparound or an exception, silent promotion of an expression to an object of a different type.

For languages with weak type checking or in situations where it is necessary to statically determine if expression results and all partial expression results will be within the range of the target type or within the range of the base type of any partials.

SM 029 Silent type conversions

As a strongly-typed language, Ada does not permit silent conversion between any types except subtypes derived from the same base type. This typing effectively forbids the uncontrolled use or inapppropriate pairing of types and operations that do not match the type. The exception for Ada is Modular Types which permit bit-wise Boolean operations on objects of these types.

More weakly typed languages can permit an object to be silently accessed as an object of a different type (eg performing boolean operations on integers or characters). This lack of separation makes the static analysis of applications hard. Additional annotations will likely be required to make this computation possible.

SM 030 Modular Types

Modular types have the traditional integer operations of integers, but have wrap-around semantics and permit bit-wise operations. Using any these operations prevents reasoning about order or the range of any object of these types. HI programs that use these operations in expressions with objects of these types must resort to dynamic checks of the final result for correct ranges when booleans are used and must dynamically verify that all input objects are within the correct ranges to prevent potential overflow before the expression is executed.

Languages with wraparound semantics on integer types and permit boolean or bitwise Boolean operations on integers have the same issues as Ada's modular types and must take the same precautions listed above for all integer operations. It is advisable that boolean operations on integer types be severly constrained to modules with appropriate analysis or banned completely.

SM 031 Fixed Point Types

Fixed point types in Ada represent a way to perform integer-based arithmetic on real numbers. The default representation of such numbers is to use the closest binary representation of the smallest number representable. For example

```
      type One_Seventh is delta 1/7 range -100.0..100.0;
```

will represent 1/7 as 1(binary), 2/7 as 10(binary), 3/7 as 11(binary), 4/7 as 101(binary), and 1.0 as 1000(binary).

Another representation is available in which 1.0 would be represented as 111(binary). This representation provides exact arithmetic but care must be taken in conversion between numbers with different representations.

The use of such numbers lets programs perform real number calculations as scaled integers while hiding the explicit scaling and eliminates problems in floating point numbers for some types of calculations.

Other languages that do not provide such a type can create scaled integers and hide the details inside appropriate modules. If scaled integers are used, strategies to handle the issues raised above, as well as separating objects of this type from other integers will be required.

Floating Point Types

SM 045 - 049

Not addressed yet

5.9 Low Level

SM 050 Explicit Control of Low Level Mechanisms

Low Level routines are those designed to explicitly control aspects of the execution environment that support the running program, such as object size and layout, bit patterns associated with data, volatility or sharing of objects.

Stronly typed langagues hide such details from the program and force explicit syntax to perform such access. For these languages, checking that such techniques are not used is almost trivial.

Weakly typed languages also have explicit mechanisms, but these mechanisms are almost regarded as part of the normal environment (for example pointer arithmetic or bitwise boolean operators).

Such mechanisms effectively prevent static analysis of the program from being done, make any kind of reasoning analysis very difficult, and make the program unportable. While many HI programs have a few places where such low level mechanisms are required, it is fundamental the these places be restriced and bounded to those places where it is mandatory and banned from elsewhere. External tools will be required to ensure that rules are enforced, and places wherethey are used excluded from program static analysis.

5.9 Interfacing

Not Done Yet

5.9 unchecked Conversion UNFINISHED

In a language with strong type checking, this issue shows up either when deliberate untype-checked conversions are applied to an object, or when runtime bounds checking is disabled. In languages without strong typechecking, the issues of this section apply everywhere throughout the program.

In general, objects of one type should only be directly converted to types which have identical representations.

5.9 Access to Subprograms

Ada's access-to-subprogram capability is intimately tied to the strong typing systems.

5.10 Generics or Templates

5.11 Pointers or Access Types

SM 080 Dynamic Memory

Dynamic memory is memory which is not assigned to any variable before the start of the main program, but which becomes assigned to an objects at some point after, and possibly is disconnected from that variable at some later point and possibly connected to another variable later. There are two basic kinds of dynamic memory, stack and heap.

SM 081 Stack Memory

Since stack memory is used to support the dynamic call chain and allocation of local storage, the major issue for HI programs is that one can statically show that stack usage is bounded and that the upper bound is less than the space allocated for the program stack. In a strongly typed language where allocated space depends upon static properties of the program, there exist static (though possibly computationally hard) algorithms to evaluat the stack requirements. In other cases, additional help such as formal annotations are probably required for this verification.

SM 082 Heap Memory and Access Types (pointers)

Heap memory is problematic for HI programs. The first issue is that all such memory is accessed through pointers, and there is substantial risk that memory used in this way will be accessed by multiple objects (aliased). It is even possible that such memory will be returned by one pointer but still referenced by another.

SM 083 Dynamic Memory Allocation

Memory that can be explicitly allocated and deallocated may be reallocated with some other base type, and if not completely initialized could be used to carry information covertly between program parts. It can also result in dangling access from uncleared pointers which now point to illegal objects.

SM 084 Space Reclamation

Often the recovery of space does not match program unit termination, and it is hard to show that allocated memory is ever released. This can result in memory leaks and possibly exhaustion of memory.

SM 085 Heap fragmentation.

Repeated allocation and deallocation of disparate types or memory amounts can lead to fragmented memory, resulting in failed allocations, even when there is enough totaol space, because insufficient contiguous space exists.