SC22/WG14/N1331

August 6, 2008

Reply to: Thomas Plum, tplum@plumhall.com

Critical Undefined Behavior

At the Delft meeting, paper N1278 was discussed and the general sentiment requested a revision of that paper. Since then, discussion on the reflector has clarified several issues, and highlighted the need to refine basic concepts. I'm going to attempt an overview summary again.

I'll use the word "bug" for a software error that causes an incorrect result to be produced. There's a discipline ("cause-effect analysis", I think) that graphs the dependence of downstream results on prior results; in this sense, a bug in some component can lead to incorrect results in all downstream components.

A component may be a critical component; a security example might be password authentication; in safety-critical code, most or all of the code may be critical components. The point here is that *any* bug in a critical component might cause a vulnerability or a hazard.

However, my main concern is with the non-critical components. There is a small set (I *think* it's small) of undefined behaviors which if they happen, even in a non-critical component, can directly cause a vulnerability or hazard. The cause-effect analysis described above assumes that each component modifies only certain objects – colloquially, it modifies its *outputs*, and if it's buggy, it may produce buggy outputs. The outputs are the values and objects which are produced or modified by the statements in the C program. But there are other values and objects that aren't meant to be directly accessed by the program statements, they're meant to be manipulated only by the system itself – such things as bookkeeping data in the heap, or a function return pointer, or the stack-frame layout in general. The distinction I'm after is that the *critical* undefined behavior category includes those that might modify this system data. Once a critical u.b. takes place, cause-effect analysis is (almost?) useless; the set of possible "downstream" effects is unbounded, or "all bets are off".

The "poster child" for critical control-flow u.b. is invocation via a pointer-to-function which has been hijacked by an attacker. For another example, system integrity can't be guaranteed if a caller assumes the wrong type for the called function. Those cases seem reasonably clear. But at Delft, Rich Peterson suggested that my "critical" category was too broad, in the area of program control flow, and I think that's correct. Joseph Myers posted some really interesting examples and questions on the reflector. I suspect that merely taking a "flaky" path through statements within one block (caused perhaps by buggy, or indeterminate, data) shouldn't be in the same "critical" category.

Doug Gwyn pointed out that if a fetch of a “trap representation” creates undefined behavior, that behavior might not actually be a trap, but just the loading of an unexpected value. (In several places, the standard refers to “explicit trap” or “produce a trap” without further definition; I used “perform a trap” to mean the same, intuitive thing.)

Discussion after Delft suggested that we don’t need to change any “undefined behavior” instances into compile-time errors; tools for the security and safety markets are already entitled to diagnose those instances severely as-is; see N1309.

This version of this paper attempts to reflect the comments by Gwyn and Myers in email messages 11497-11500; details to be provided at the next meeting. In particular, note that fetching an indeterminate value is permitted to perform a trap, but failing that, the value produced is not required to be deterministic (e.g., an indeterminate **bool** might be sometimes true and sometimes false).

Note also that various other undefined behaviors that produce incorrect pointer values are only one step away from an out-of-bounds store; still we maintain the distinction that the out-of-bounds store is the critical undefined behavior. In other words, an incorrect pointer value, that never got used, never actually produced a critical undefined behavior.

Proposed Wording

Replace

> 3.4.3 **undefined behavior**
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

as follows (re-numbering as needed):

> 3.4.3a **out-of-bounds store**
> an (attempted) access (3.4.1) which, at run-time, for a given computational state, would modify one or more bytes (or for an object declared **volatile,** would fetch one or more bytes) that lie outside the bounds permitted by the standard
>
> 3.4.3b **improper control flow**
> an (attempted) alteration of the flow of control which would violate the semantics specified by the standard [needs work] or invoke a function which is not compatible with the type of the invoking expression; however, performing a trap is not an improper control flow
>
> 3.4.3c **undefined behavior**
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements, except that the behavior shall not perform an out-of-bounds store or improper control flow

3.4.3d **critical undefined behavior**
behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE the behavior might perform an out-of-bounds store or an improper control flow

Identify the following undefined behavior situations as **critical undefined behavior:**

6.2.4 An object is referred to outside of its lifetime
6.3.2.1 An lvalue does not designate an object when evaluated
6.3.2.3 A pointer is used to call a function whose type is not compatible with the pointed-to type
6.5.2.2 A function is defined with a type that is not compatible with the type (of the expression) point to by the expression that denotes the called function
6.5.2.2 For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters
6.5.2.2 For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion
6.5.2.2 For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters
6.5.3.2 The operand of the unary * operator has an invalid value
6.5.6 Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated
7.1.4 An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments
7.13.2.1 The `longjmp` function is invoked to restore a nonexistent environment
7.20.3 The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used
7.20.4.3 During the call to a function registered with the `atexit` function, a call is made to the `longjmp` function that would terminate the call to the registered function
7.21.1, c7.24.4 A string or wide string utility function is instructed to access an array beyond the end of an object