"Poor code quality leads to unpredictable behavior. From a user's perspective that often manifests itself as poor usability. For an attacker it provides an opportunity to stress the system in unexpected ways."

— Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors (Tsipenyuk, Chess, McGraw)

Integer Safety in C and z/TPF

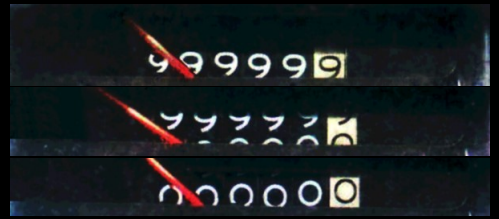
Ian S. Worthington

Société nationale des chemins de fer français

May 2022

1. This deck best displayed with the Pympress dual screen PDF presentation tool, or similar.
2. As the size of our processed data grows, so the limitations and danger of the lack of integer safety in C become apparent.
3. This is *not* a z/TPF problem, but a designed-in problem with C.

Wraparound



1. A real world example of numeric wraparound.

Why do we care?

- Therac 25 radiation therapy — deaths
- Ariane 5 — launch explosion, cost: $370M.
- Delta/Comair — crew rescheduling. Cancelled 3900 flights, 200k pax. Cost: $20M.
- 787 Dreamliner — periodic power off/on.

Lack of integer safety has real word consequences, both financial and life-altering.

1. Between 1985–87 software errors, including arithmetic overflow, caused massive overdoses of radiation to at least 6 patients, 3 of which died.
2. On 4 June 1996 the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded, resulting in the most expensive software bug in history at a cost of $370M. This was caused by code without protection against integer overflow.
3. In Dec 2004 a severe winter storm hit the American midwest resulting in the cancellation of postponement of 91% of Comair's flights. The crew management system could only handle 32 000 changes per month and crashed resulting in the cancellation of 3900 flights and the stranding of 200 000 passengers, at a cost of $20M: the entire profit from the previous quarter. The Comair President was ultimately to lose his job.
4. In 2015 the FAA and the EASA instructed 787 operators to periodically reset its electrical system to prevent lost of power and ram air turbine deployment as a result of integer overflow which would otherwise occur ever 248 days.

Does C care?

- *C11 6.5/5: If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behaviour is undefined.*
- **Translation**: The results of overflow/wraparound are undefined.

1. What does the C standard have to say about integer safety?

Does C care? Unsigned integers are special

- *C11 6.2.5/9*: *The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same. A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.*

- **Translation**: Unsigned integer types are special case and perform MODULO arithmetic by design.

1. No, I don't expect you to read this

Does C care? Summary

- Unsigned integers: reduced modulo MAX(type).
- Signed integers: undefined.
- Nothing is an error.

1. The C standard was built upon the then-existing compiler implementations of Kernighan and Ritchie's 1970s-era C specification, warts and all.

Impacts: CVE database

"Common Vulnerabilities and Exposures" Over 3000 recorded cases of integer issues.

Availability

Undefined behaviour, crashes, infinite loops, DoS

Integrity

Data corruption

Confidentiality/Availability/Access Control

Bypass protection (probably not under TPF?)

1. The Common Vulnerabilities and Exposures database, funded by the US government, attempts to categorise problems with software products.
2. There are a number of flavours of "Numeric Issues (CWE-189)" in the CVE database. This is from CWE-190: "Integer Overflow or Wraparound"
3. Availability: "DoS: Crash, Exit, or Restart; DoS: Resource Consumption (CPU); DoS: Resource Consumption (Memory); DoS: Instability". This weakness will generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.
4. Integrity: "Modify Memory". If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the wrap around results in other conditions such as buffer overflows, further memory corruption may occur.
5. C/A/AC: "Execute Unauthorized Code or Commands; Bypass Protection Mechanism". This weakness can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.

Does this affect SNCF?

Yes!

Dec ’21 – Apr ’22 XML/JSON parsing infinite loops and subsequent months of forced manual processing of large group PNRs caused by integer overflow processing large B2B documents.

Refresher: Unsigned types

	Unsigned Range	
Width	min	max
8	0	2^8 - 1 FF_{16} 255_{10}

Refresher: Unsigned types

Width	Unsigned Range min	Unsigned Range max
8	0	$2^8 - 1$ FF_{16} 255_{10}
16	0	$2^{16} - 1$ $FFFF_{16}$ $65\,535_{10}$
32	0	$2^{32} - 1$ $FFFF\ FFFF_{16}$ $4\,294\,967\,295_{10}$
64	0	$2^{64} - 1$ $FFFF\ FFFF\ FFFF\ FFFF_{16}$ $18\,446\,744\,073\,709\,551\,615_{10}$

Refresher: Signed types

Width	Signed Range (two's complement)	
	min	max
8	$-(2^7)$ 80_{16} -128_{10}	$2^7 - 1$ $7F_{16}$ 127_{10}

Refresher: Signed types

Width	Signed Range (two's complement)	
	min	max
8	$-(2^7)$ 80_{16} -128_{10}	$2^7 - 1$ $7F_{16}$ 127_{10}
16	$-(2^{15})$ 8000_{16} $-32\ 768_{10}$	$2^{15} - 1$ $7FFF_{16}$ $32\ 767_{10}$
32	$-(2^{31})$ $8000\ 0000_{16}$ $-2\ 147\ 483\ 648_{10}$	$2^{31} - 1$ $7FFF\ FFFF_{16}$ $2\ 147\ 483\ 647_{10}$
64	$-(2^{63})$ $8000\ 0000\ 0000\ 0000_{16}$ $-9\ 223\ 372\ 036\ 854\ 775\ 808_{10}$	$2^{63} - 1$ $7FFF\ FFFF\ FFFF\ FFFF_{16}$ $9\ 223\ 372\ 036\ 854\ 775\ 807_{10}$

A demonstration: define some maximum values

"Undefined behaviour": What happens under z/TPF?

```
__int8_t   s8  = 0x7f;               // char
__int16_t s16  = 0x7fff;             // short
__int32_t s32  = 0x7fffffff;         // int
__int64_t s64  = 0x7fffffffffffffff; // long

__uint8_t  u8  = 0xff;
__uint16_t u16 = 0xffff;
__uint32_t u32 = 0xffffffff;
__uint64_t u64 = 0xffffffffffffffff;
```


1. Really, of course, this is "What happens on IBM mainframes", as the behaviour is dictated and described by the Principles of Operations.

A demonstration: increment them


```
sprintf(b, "max s8: %4i, s16: %6hi, s32: %11i, s64: %20li",
  s8, s16, s32, s64); wtopc_text(b);

s8++; s16++; s32++; s64++;

sprintf(b, "p1  s8: %4i, s16: %6hi, s32: %11i, s64: %20li",
  s8, s16, s32, s64); wtopc_text(b);
```


1. Only signed shown, same for unsigned

A demonstration: results

```
max s8:  127, s16:  32767, s32:  2147483647, s64:  9223372036854775807
p1  s8: -128, s16: -32768, s32: -2147483648, s64: -9223372036854775808

max u8:  255, u16:  65535, u32:  4294967295, u64: 18446744073709551615
p1  u8:    0, u16:      0, u32:           0, u64:                    0
```

As promised:

No errors generated!

1. Unsigned integers wrap modulo n, as demanded by the C specification; Signed integers wrap from their maximum value to their minimum.

OK, so it's a problem: Give me a solution

- Partial solution attempts in various compilers
- Maybe something basic in C2x standard?
- C2x prototypes seen to date are slow
- Need something now, robust, efficient

1. A number of attempts at solutions have been implemented in various compilers with varying levels of success. There may be something incorporated into the next C2x standard. But we need a solution now and we need that solution to be both robust and fast, not dependent upon slow post-operation checks. The IBM machine architecture has always provided safe arithmetic operations, so let's build a solution around that.

API

- "Standard" API

```
inline extern __attribute__((always_inline))
bool ckd_add_u64( __uint64_t *u64sum,
                  __uint64_t u64a,
                  __uint64_t u64b,
                  bool       abort )
```


1. This structure of this api is similar to those proposed for the C2x standard and previous prototypes
2. Maybe a bit non-intuitive? Let's walk through it.
3. inline: suggestion to the compiler that this function should be inlined
4. extern: prevent the generation of a callable function
5. always_inline: force inlining even if noinline specified
6. __uint*NN*_t: preferred to types such as short and long as the type size is made explicit.
7. u64a, u64b: addends
8. u64sum: sum
9. abort: SERRC on overrflow or return?
10. return: TRUE if overflow

API

- "Standard" API

```
inline extern __attribute__((always_inline))
bool ckd_add_u64( __uint64_t *u64sum,
                  __uint64_t u64a,
                  __uint64_t u64b,
                  bool       abort )
```

- Easy to use

- Focus on error flow first, then normal flow

```
#include "checked_integer_arithmetic.inl"

if ckd_add_u64(&sum, a, b, false) {
        // handle overflow elegantly
}
// carry on with processing
```


1. This structure of this api is similar to those proposed for the C2x standard and previous prototypes
2. Maybe a bit non-intuitive? Let's walk through it.
3. inline: suggestion to the compiler that this function should be inlined
4. extern: prevent the generation of a callable function
5. always_inline: force inlining even if noinline specified
6. __uint*NN*_t: preferred to types such as short and long as the type size is made explicit.
7. u64a, u64b: addends
8. u64sum: sum
9. abort: SERRC on overrflow or return?
10. return: TRUE if overflow

Scope

- 12 function calls
  - Addition, Subtraction, Multiplication, Division, Remainder
  - 64, 32, 16 bit
- SERRC type selectable via #define
- Debug tracing selectable via #define

1. Division: Quotient cannot exceed dividend, so no overflow possible. This function then provided for 1: consistency, 2: divide by zero trapping, 3: future expansion to unmatched types, 4: obtain both quotient and remainder for the cost of a single division.

Restrictions

- Unsigned integers only: focus is on fixing pointer problems. (Signed can be provided.)
- Matching types only: focus as above. (Non-matching can be provided.)
- Not infix: C language restriction. (C++ interface using exceptions can be provided.)

Efficient solutions

- What we'd like to code:

```
algr ...      add
brc 3,toobig  branch on carry set
xr ...        set successful
```


1. The IBM machine architecture has always provided safe arithmetic operations, so let's build a solution around that.
2. Unstable due to potential optimiser reordering of statements. Might be possible in gcc v11+, but this not yet supported by TPF.
3. 16 bit types so not have hardware detection of overflow so post-operation checks are implemented instead.

Efficient solutions

- What we'd like to code:

```
algr ...      add
brc 3,toobig  branch on carry set
xr ...        set successful
```

- Not quite possible/unsafe in gcc < v11

- What we do instead:

```
algr ...      add
ipm ...       get cc
nilf ...      test carry bit/set sucessful
jnz toobig    branch on carry set
```

- Almost as good — one additional (fast) instruction

1. The IBM machine architecture has always provided safe arithmetic operations, so let's build a solution around that.
2. Unstable due to potential optimiser reordering of statements. Might be possible in gcc v11+, but this not yet supported by TPF.
3. 16 bit types so not have hardware detection of overflow so post-operation checks are implemented instead.

Further reading

- David Svoboda – Towards Integer Safety (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2428.pdf)

- Tsipenyuk, Chess, McGraw — Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors (https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%
%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-
%20Chess%20-%20McGraw.pdf)

- Common Weakness Enumeration: Numeric Errors (https://cwe.mitre.org/data/definitions/189.html)

- C11 final draft (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf)