Prevalence of Integer Wraparound in C/C++

Will Dietz
University of Illinois at Urbana-Champaign
wdietz2@illinois.edu

Vikram Adve
University of Illinois at Urbana-Champaign
vadve@illinois.edu

ABSTRACT

Integer wraparound is a well-known source of security vulnerabilities in C and C++ programs. However integer wraparound are not always security concerns, and what remains an open question is the frequency these *benign* wraparounds occur in C and C++ programs. To answer this question we created a tool to instrument C and C++ codes to check for integer wraparound. We ran this tool on the Olden and SPEC CINT2000 benchmark suites, and found 74 static sources of wraparound across 8 of the 22 benchmarks evaluated. We analyzed each of these and present our findings in this report.

1. INTRODUCTION

Integer wraparound as a source of security vulnerabilities is an established concept. An unexpected integer wraparound in an allocation size calculation or a sensitive control flow decision can lead to any number of security concerns. Security vulnerabilities are especially common in programs written in languages such as C and C++. However it is erroneous to consider all integer wraparounds security concerns, as many have no security implications whatsoever.

The C language specification defines the behavior of unsigned integer wraparound, but signed integer wraparound is explicitly undefined. However both types can either lead to security vulnerabilities, or be entirely harmless. In both cases, the wraparound is not an error by itself; it isn't the overflowed operation that is an error but rather how it's used. We call these wraparounds that have no security implications *benign*.

In the context of building a security tool, the existence of benign wraparounds presents a complication. While some wraparounds result in security errors, disallowing benign wraparound would break many otherwise valid programs. Additionally, these wraparounds occur using both defined and undefined behavior, suggesting one cannot simply reject programs using undefined behavior to handle the security ramifications of integer wraparound while still allowing valid programs.

To motivate this argument, and to help explore how common these benign wraparounds are in C/C++ programs, we built Overflow Check Inserter (OCI), a tool for instrumenting codes to check for signed and unsigned integer wraparounds (both positive and negative overflows). We ran our tool on programs from the Olden and SPEC CINT2000 benchmark suites, and analyzed the results.

In the analysis of our experiments, we found that there are a number of common uses of wraparound in C programs that are benign. The full set is shown in Table 1, but common examples include hash function implementations, random number generation, memory allocation and management logic, and cheap modulo calculation. These examples and others are legitimate uses of integer wraparound and are not security errors. Furthermore, programs such as the 254.gap benchmark we analyzed explicitly use signed wraparound semantics (which are technically undefined in C), but are also benign and have no security implications.

The rest of this report is structured as follows: in Section 2 we present our tool, and in Section 3 we walk through representative examples of wraparound in detail, as well as highlight some of the more interesting static sources of wraparound behavior. A summary of our findings is shown in Table 1 at the end of the report.

2. TOOL DESIGN

In this section we describe "Overflow Check Inserter" (OCI), our tool for instrumenting C and C++ programs to detect integer wraparounds (both positive and negative overflow).

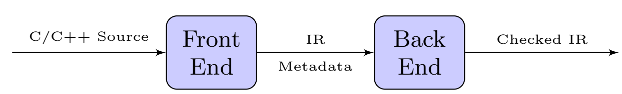

Figure 1: OCI System Design

OCI is built as a part of our ongoing work on SVA[1], and uses both the LLVM[3] compiler infrastructure and its C/C++ front-end Clang[2]. This has two significant effects on OCI. First, OCI is the first iteration of an extension to SVA to support handling of numerical errors. As such, some of the design decisions are targeted towards building a more general tool as opposed to simply detecting integer wraparound. Second, the LLVM IR does not distinguish between signed and unsigned integers, which poses a problem when adding checks for wraparound[1].

Figure 1 depicts the resulting two component design of OCI:

[1]What it means for an integer to wraparound is different at the bit level for signed vs unsigned integers. Consider 1111+0001 as two 4-bit integers, whose sum we know is 0000. However if these are unsigned, we wrapped around from 15 to 0. If signed, we went from -1 to 0, not a wraparound.

the annotation front-end and the backend-end transformation pass, and we discuss each below.

2.1 Front-End

The front-end part of OCI is implemented as a modification to the C/C++ LLVM front-end, Clang. During code-generation (Abstract Syntax Tree (AST) to LLVM IR transformation), we use source-level information (encoded in the AST) to extend the generated IR to distinguish between arithmetic operations that produce signed and unsigned integers. We store this information as annotations on the generated IR.

2.2 Transformation

Once the IR has the metadata to distinguish signed and unsigned values, OCI has an LLVM pass that walks through the program looking for integer arithmetic operations. For each suitable operation, we replace it with code that checks for wraparound and when a wraparound instance occurs either calls abort() or a user-specified function. We also have a debug version that logs each error, including information such as the filename, line number, type of operation, the signedness of the operation, and the dynamic values that resulted in the error. This debug information was used for the experiments presented in Section 3.

We check for wraparound by making use of LLVM intrinsics, which on x86 get lowered to the appropriate ALU flag checks, making for minimal additional code and very fast checks due to highly predictable branches.

The operations supported by OCI are addition, subtraction, and multiplication. We ignore division (which only wraps around on INT_MIN/-1) because it is a very rare edge case, and expensive to check for. We don't check bitwise operations such as and, xor, or shift. Our current implementation also does not check for casting errors such as truncation.

Once the transformation phase is complete, OCI no longer needs to maintain the signedness metadata, and we are free to run LLVM optimizations to improve code quality. This has the advantage of enabling us to optimize the instrumented code, but at the risk of not checking any operations introduced by the compiler. We are interested only in wraparounds originating from the source.

3. ANALYSIS OF WRAPAROUND SOURCES

In this section we look at representative examples from the set of static wraparounds we found, in an attempt to reflect and capture the frequency with which wraparound errors occur and the most common techniques that result in benign wraparounds.

3.1 Olden

The Olden[4] set of benchmarks has 10 small programs (between 245 and 2,073 lines of code each), 2 of which had instances of integer wraparound. We look at both of these below.

3.1.1 *bisort*

The bisort program from Olden is a good example of benign wraparound. As shown in Listing 1 bisort uses unsigned char to provide a cheap way to compute modulo 256 computation in a counter. This is benign because the wraparound is intentional and the result is only used in a safe way.

Listing 1: Wraparound in Olden's bisort

```
void InOrder(HANDLE *h) {
  HANDLE *l, *r;
  if ((h != NIL)) {
    l = h->left;
    r = h->right;
    InOrder(l);
    static unsigned char counter = 0;
    if (counter++ == 0)    /* reduce IO */
      printf("%d @ 0x%x\n",h->value, 0);
    InOrder(r);
  }
}
```

3.1.2 *perimeter*

Olden's perimeter benchmark is a good example of potentially benign wraparound, but with signed integers. Listing 2 shows the source of a signed wraparound in the multiplication on the indicated line. Looking at the code it appears the programmer did not anticipate the possibility of that computation wrapping around. Not only that, but because it uses signed integers it is relying on undefined behavior. Despite both of these, it is still unclear whether or not this wraparound should be considered benign. To fully determine if this is benign, we would have to evaluate its effects upon the rest of the program in the context of a particular security policy.

Listing 2: Wraparound in Olden's perimeter

```
static int CheckOutside(int x, int y)
{
  int euclid = x*x+y*y;

  if (euclid > 4194304) return 1;
  if (euclid < 1048576) return -1;
  return 0;
}
```

3.2 SPEC CINT2000

The second set of benchmarks we investigated was the SPEC 2000 integer benchmark suite (CINT2000). This suite consists of 12 medium-sized programs (2,412 to 222,210 lines of code), 6 of which used integer wraparound in executions using their intended data sets. We analyze some of these wraparound errors below.

3.2.1 *175.vpr*

The 175.vpr benchmark had two wraparounds here, and they're almost identical, Listing 3 shows one of them. The other is in 'my_frand', and does something similar to create a random floating pointer number. Both instances comment on their intentional use of unsigned integer wraparound. This is a benign wraparound, using wraparound semantics in the intended way.

3.2.2 *176.gcc*

176.gcc had quite a few wraparounds–our experiments found 24 static sources of wraparound. Causes vary from bit

Listing 3: Intentional use of wraparound in a random number generator

```c
/* Portable random number generator defined below.  Taken from ANSI C by  *
 * K & R.  Not a great generator, but fast, and good enough for my needs. */

#define IA 1103515245u
#define IC 12345u
#define IM 2147483648u
#define CHECK_RAND

static unsigned int current_random = 0;

int my_irand (int imax) {

  /* Creates a random integer between 0 and imax, inclusive.  i.e. [0..imax] */

  int ival;

  /* current_random = (current_random * IA + IC) % IM; */
  current_random = current_random * IA + IC;  /* Use overflow to wrap */
  ival = current_random & (IM - 1);  /* Modulus */
  ival = (int) ((float) ival * (float) (imax + 0.999) / (float) IM);

#ifdef CHECK_RAND
  if ((ival < 0) || (ival > imax)) {
    printf("Bad value in my_irand, imax = %d  ival = %d\n",imax,ival);
    exit(1);
  }
#endif

  return(ival);
}
```

manipulation, to hashing (on signed integers), to unary negation on INT_MIN, to trying to find the largest representable signed integer. We discuss the some interesting sources below.

Listing 4: Use of wraparound to get largest representable signed int

```c
/* (unsigned) <= 0x7fffffff is equivalent to >= 0. */
else if (const_op == ((HOST_WIDE_INT) 1 << (mode_width - 1)) - 1)
{
  const_op = 0, op1 = const0_rtx;
  code = GE;
}
break;
```

Listing 4 shows an example where the code is trying to compute the largest representable signed integer (HOST_WIDE_INT is simply an 'int', and mode_width is 32). Note that this code ends up computing 2^{31} and then casting it as a signed integer. In two's complement, the casted result is -2^{31}, which they then subtract one from causing the value to wraparound to INT_MAX ($2^{31} - 1$). This appears to be a programming error as there's no reason to cast to a signed int in the calculation–it would be equivalent and have no wraparound (signed or otherwise) if the entire computation was done as unsigned and then cast the result to signed.

Listing 5 shows code that attempts to determine properties about the integer passed in at a bit level. In doing so, it invokes various arithmetic operations (subtraction, addition, and another subtraction in the POWER_OF_2_or_0 macro) that wraparound. These are all on unsigned integers and are carefully constructed to test the correct bits in the integers, so all of these wraparounds are benign.

Listing 5: Wraparound in bit manipulation operations

```c
#define POWER_OF_2_or_0(I) (((I) & ((unsigned)(I) - 1)) == 0)

int
integer_ok_for_set (value)
     register unsigned value;
{
  /* All the "one" bits must be contiguous.  If so, MASK + 1 will be
     a power of two or zero.  */
  register unsigned mask = (value | (value - 1));
  return (value && POWER_OF_2_or_0 (mask + 1));
}
```

Listing 6: Casting combined with arithmetic results in wraparound used in allocation

```c
/* Allocate an rtx vector of N elements.
   Store the length, and initialize all elements to zero.  */

rtvec
rtvec_alloc (n)
     int n;
{
  rtvec rt;
  int i;

  rt = (rtvec) obstack_alloc (rtl_obstack,
                              sizeof (struct rtvec_def)
                              + (( n - 1) * sizeof (rtunion)));

  /* clear out the vector */
  PUT_NUM_ELEM(rt, n);
  for (i=0; i < n; i++)
    rt->elem[i].rtvec = NULL;   /* @@ not portable due to rtunion */

  return rt;
}
```

In Listing 6 we see an allocation wrapper function that allocates a vector of n elements. It starts with 16 bytes and then adds $(n-1) * 8$ more to fill out the array, since the beginning rtvec_def struct has room for 1 by default. Due to integer promotion rules and because *sizeof* returns an unsigned value, there's an implicit cast of $(n-1)$ to unsigned before it's multiplied with the sizeof value, and the result of the multiplication (which is unsigned) is added to the first sizeof in the computation. Now consider the case when $n = 0$. Then $(n-1)$ evaluates to -1 which when cast to an unsigned integer is UINT_MAX, 2^{32}. Multiplying this by the sizeof value(8) then causes a wraparound to 8 bytes. Note that this is the correct allocation size (size of the base struct, without it's builtin single element, since $n = 0$ means we don't want any elements).

It's difficult to categorize this wraparound instance as benign or not. The expression when $n = 0$ evaluates to 8 bytes, which means it allocates all of the 'rtvec_def' structure, minus the hardcoded single element. As long as no code tries to access that element (or copy the structure by value), the code does what the comments suggest it does: allocates an rtx vector of the specified number of elements. However intentionally under-allocating a struct (and using wraparound to get there) highlights how complex it can be to classify wraparounds as benign or not. In this particular case we categorize it as benign because the resulting struct is used safely in the rest of the program.

3.2.3 186.crafty

In Listing 7 the code is attempting to determine if the unsigned variables WhiteBishops and BlackBishops are powers

Listing 7: Determining if an integer is a power of two

```
/*
 --------------------------------------------------------
|                                                        |
|   now, give either side a bonus for having two bishops.  |
|                                                        |
 --------------------------------------------------------
*/
if (And(WhiteBishops,WhiteBishops-1)) score+=BISHOP_PAIR;
if (And(BlackBishops,BlackBishops-1)) score-=BISHOP_PAIR;
```

of two through bit manipulation. The wraparound occurs on the subtraction when either variable is zero, resulting in an unsigned wrap-around error to UINT_MAX. This is benign because while it isn't clear from the code that the wraparound is intentionally accounted for, it does seem to compute the correct result and therefore is using the defined wraparound rules in a safe way.

3.2.4 197.parser

Listing 8: Find second-largest power of two that fits in an unsigned integer

```
void initialize_memory(void) {
    SIZET i, j;
    if ((MEMORY_ALIGNMENT & (MEMORY_ALIGNMENT-1)) != 0) {
        fprintf(stderr, "sizeof(Align) is not a power of 2.\n");
        exit(1);
    }
    for (i=0, j=1; i < j; i = j, j = (2*j+1)) largest_block = i;
    largest_block &= ALIGNMENT_MASK;
    largest_block += -sizeof(Nuggie); /* must have room for a nuggie
            too */
```

197.parser had a single wraparound source, which occurred in its custom allocator. As shown in Listing 8, the allocator uses unsigned wraparound intentionally to compute information about representable powers of two.

3.2.5 254.gap

254.gap is a benchmark that intentionally uses signed arithmetic in its computations. From the LLVM developer's mailing list[2]:

> This benchmark thinks overflow of signed multiplication is well defined. Add the -fwrapv flag to ensure that the compiler thinks so too.

We did not investigate the errors in this benchmark due to the complex and obfuscated nature of the code. However, as shown in Table 1, our tool reported many sources of signed wraparound as expected.

4. CONCLUSION

We implemented a tool to check C and C++ programs for integer wraparounds and examined two benchmark suites for sources of benign wraparounds. We reported them here, and with analysis of the source of each wraparound instance. We found that a large number of programs in our set (8 of 22) used integer wraparound as part of their execution, suggesting that numerical errors have common benign uses and cannot be entirely classified as security errors by themselves.

[2] `http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20110131/115969.html`

5. REFERENCES

[1] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *SOSP '07: Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*, October 2007.

[2] C. Lattner. Clang. `http://clang.llvm.org/`.

[3] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 222–233, New York, NY, USA, 1996. ACM.

Table 1: Summary of all instances of integer wraparound reported in SPEC and Olden

Benchmark	Location	Type[1]	Op	Operand 1[2]	Operand 2[2]	Description
bisort	bitonic.c:36	U	Add	255	1	Cheap modulo 256 calculation. (Listing 1)
perimeter	maketree.c:8	S	Mul	−1003520	−1003520	Normal mult (Listing 2)
175.vpr	util.c:463	U	Mul	10000128	1103515245	Random number generator (Listing 3)
175.vpr	util.c:484	U	Mul	10000422	1103515245	Random number generator
176.gcc	combine.c:9024	S	Sub	−2147483648	1	Find largest signed int (Listing 4)
176.gcc	combine.c:9057	S	Sub	−2147483648	1	Find largest signed int
176.gcc	combine.c:9572	S	Sub	−2147483648	1	Find largest signed int
176.gcc	cse.c:745	S	Sub	0	−2147583648	Signed negation on INT_MIN
176.gcc	cse.c:1905	U	Add	4294961100	12983	Hash
176.gcc	cse.c:1912	U	Add	6016	4294963199	Hash
176.gcc	m88k.c:85	S	Sub	0	−2147583648	Signed negation on INT_MIN
176.gcc	m88k.c:127	U	Sub	0	1	Bit manipulation (Listing 5)
176.gcc	m88k.c:128	U	Sub	0	1	Bit manipulation
176.gcc	m88k.c:128	U	Add	4294967295	1	Bit manipulation
176.gcc	m88k.c:887	U	Add	4294903640	65535	Straightforward add
176.gcc	m88k.c:1349	U	Add	4294967195	65535	Straightforward add
176.gcc	m88k.c:2133	U	Add	4294966880	65535	Straightforward add
176.gcc	obstack.c:271	U	Sub	0	1	Signed/Unsigned type promotion
176.gcc	recog.c:1847	S	Sub	0	−2147583648	Signed negation on INT_MIN
176.gcc	regclass.c:1228	S	Sub	0	−2147583648	Signed negation on INT_MIN
176.gcc	reload.c:2962	S	Sub	0	−2147583648	Signed negation on INT_MIN
176.gcc	stor-layout.c:1039	S	Sub	−2147583648	1	Find largest signed int
176.gcc	rtl.c:193	U	Mul	4294967295	8	Allocation calculation (Listing 6)
176.gcc	rtl.c:193	U	Add	16	4294967288	Allocation calculation
176.gcc	rtl.c:215	U	Mul	4294967295	8	Allocation calculation
176.gcc	rtl.c:215	U	Add	16	4294967288	Allocation calculation
176.gcc	tree.c:1222	S	Mul	1001175137	613	Hash
176.gcc	varasm.c:2255	S	Mul	10067065	613	Hash
186.crafty	evaluate.c:594	U	Sub	0	1	Bit manipulation (Listing 7)
186.crafty	evaluate.c:595	U	Sub	0	1	Bit manipulation
186.crafty	utility.c:813	U	Add	1005223449	4078606337	Random number generation
197.parser	and.c:365	U	Add	2489845425	2066441660	Hash
197.parser	and.c:367	U	Add	2368899675	2065686955	Hash
197.parser	and.c:397	U	Add	2211902768	2120440914	Hash
197.parser	fast-match.c:101	U	Add	2437185530	2120440914	Hash
197.parser	parse.c:167	U	Add	2167639695	2133829292	Hash
197.parser	parse.c:168	U	Add	2161008554	2134005536	Hash
197.parser	parse.c:169	U	Add	2162098460	2133829292	Hash
197.parser	prune.c:161	U	Add	2257027530	2066441660	Hash
197.parser	prune.c:357	U	Add	2161771251	2133829292	Hash
197.parser	prune.c:1032	U	Add	2437185530	2120440914	Hash
197.parser	xalloc.c:68	U	Mul	2	4294967295	Find largest representable square (Listing 8)
197.parser	xalloc.c:70	U	Add	2147483640	4294967292	Same allocation code as above.
253.perlbmk	hv.c:138	U	Mul	1041992573	33	Hash
253.perlbmk	hv.c:229	U	Mul	1000000857	33	Hash
253.perlbmk	hv.c:308	U	Mul	1004113817	33	Hash
254.gap	idents.c:151	U	Mul	65599	1000107457	Hash
254.gap	idents.c:192	U	Mul	65599	1000107457	Hash
254.gap	idents.c:241	U	Mul	65599	1000271814	Hash
254.gap	idents.c:276	U	Mul	65599	1003143503	Hash
254.gap	integer.c:526	S	Mul	65536	33791	Intentional Signed Overflow
254.gap	integer.c:529	S	Mul	65536	33540	Intentional Signed Overflow
254.gap	integer.c:611	S	Mul	65536	36085	Intentional Signed Overflow
254.gap	integer.c:614	S	Mul	65536	33540	Intentional Signed Overflow
254.gap	integer.c:661	S	Mul	1000859064	8	Intentional Signed Overflow
254.gap	integer.c:681	S	Mul	37111	63169	Intentional Signed Overflow
254.gap	integer.c:744	S	Mul	33584	63978	Intentional Signed Overflow
254.gap	integer.c:745	S	Mul	33649	64016	Intentional Signed Overflow
254.gap	integer.c:746	S	Mul	33584	64747	Intentional Signed Overflow
254.gap	integer.c:747	S	Mul	33622	64182	Intentional Signed Overflow
254.gap	integer.c:804	S	Mul	33120	65057	Intentional Signed Overflow
254.gap	integer.c:805	S	Mul	33629	65172	Intentional Signed Overflow
254.gap	integer.c:806	S	Mul	34021	63646	Intentional Signed Overflow
254.gap	integer.c:807	S	Mul	33120	65021	Intentional Signed Overflow
254.gap	integer.c:1454	S	Mul	40320	53344	Intentional Signed Overflow
254.gap	integer.c:1467	S	Mul	65536	33757	Intentional Signed Overflow
254.gap	integer.c:1470	S	Mul	65536	33878	Intentional Signed Overflow
254.gap	integer.c:1527	S	Mul	65536	33386	Intentional Signed Overflow
254.gap	integer.c:1533	S	Mul	37050	58471	Intentional Signed Overflow
254.gap	integer.c:1541	S	Mul	33756	64512	Intentional Signed Overflow
254.gap	integer.c:1572	S	Mul	65536	36085	Intentional Signed Overflow
254.gap	integer.c:2039	S	Mul	65536	33386	Intentional Signed Overflow
254.gap	integer.c:2044	S	Mul	34624	63049	Intentional Signed Overflow
254.gap	integer.c:2052	S	Mul	33756	64512	Intentional Signed Overflow

[1] Type indicates whether the operation is signed or unsigned, 'S' and 'U' respectively.
[2] Operands reported are from at least one of the errors triggered at a particular site.