CSC 482/582: Computer Security

Integer Security
Topics

1. Computer Integers
2. Integers in C and Java
3. Undefined Behavior
4. Overflow Examples
5. Checking for Overflows

256th “Split Screen” level of Pac-Man
https://en.wikipedia.org/wiki/Pac-Man
December 25, 2004
Flight crew scheduling software stopped.
Cancelled all 1100 flights that day.
What happened?
Winter weather led to many crew changes.
Number of changes > 32,767
Computer Integers

Computer integers are not the same set of numbers as mathematical integers.

- Finite set, not infinite.

What happens when integer calculations result in a number outside that set?

- Set carry or overflow flag in CPU.
- Throw an exception.
- Convert integer type to higher precision.
- Saturation (remain at maximum/minimum value).
- Wrap from max to min or min to max value.

Depends on language and hardware.
Unsigned Integers

Slide #5
Two’s Complement

Two’s complement = One’s complement + 1. Sign is represented by most significant bit. Range: \(-2^{n-1}..2^{n-1}-1\), only one representation of 0.

\[
\begin{array}{cccccccc}
+75 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
\text{Comp} & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
-75 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]
Two’s Complement
Can’t Sleep

https://xkcd.com/571/
Assembly Language

CPU Carry Flag
- Check for unsigned arithmetic.
- Set if + or * exceeds maximum value.
- Set if subtraction result is beyond min value.

CPU Overflow Flag
- Check for signed arithmetic.
- Set if sum two numbers w/o sign bit, result has sign bit set.
- Set if sum two numbers with sign bit, result does not have sign bit set.

GPUs and DSPs saturate instead of wrap.
## C Integers on 64-bit Windows

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>-9.223 x 10^{18}</td>
<td>9.223 x 10^{18}</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64</td>
<td>0</td>
<td>1.844 x 10^{19}</td>
</tr>
</tbody>
</table>
Additional C Integer Types

- `size_t` is guaranteed to have sufficient precision to represent size of an object.
- `int##_t` and `uint##_t` represent integers of exactly the specified `#` of bits, e.g. `uint24_t`.
- `int_fast##_t` and `uint_fast##_t` represent integers with at least `#` bits that are fastest for CPU to use.
- `intmax_t` and `uintmax_t` are the largest machine integers available.
- Vendors often define platform specific integer types, e.g. `__int32`, `DWORD`, etc. on Windows.
- Use GNU Multiple Precision Arithmetic Library (GMP) when integers larger than machine precision are needed.
# Java Integers

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>-9.223 x 10^{18}</td>
<td>9.223 x 10^{18}</td>
</tr>
</tbody>
</table>
public static void main(String args[]) {
    long product = 1;
    for(int i = 1; i <= 21; i++) {
        System.out.print(i);
        System.out.print("! = ");
        product *= i;
        System.out.println(product);
    }
}
Output

1! = 1
2! = 2
3! = 6
....
20! = 2432902008176640000
21! = -4249290049419214848
import java.math.BigInteger;
public class BigFactorials
{
    public static void main(String args[])
    {
        BigInteger product = BigInteger.ONE;
        BigInteger index = BigInteger.ONE;

        for(int i = 1; i <= 21; i++)
        {
            System.out.print(i);
            System.out.print("! = ");
            product = product.multiply(index);
            System.out.println(product);
            index = index.add(BigInteger.ONE);
        }
    }
}
Output

1! = 1
2! = 2
3! = 6
....
20! = 2432902008176640000
21! = 51090942171709440000
### C/C++ Integer Operations

#### Examples of C/C++ Integer Operations and Their Results

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT_MAX+1</td>
<td>0</td>
</tr>
<tr>
<td>LONG_MAX+1</td>
<td>undefined</td>
</tr>
<tr>
<td>INT_MAX+1</td>
<td>undefined</td>
</tr>
<tr>
<td>SHRT_MAX+1</td>
<td>SHRT_MAX+1 if INT_MAX&gt;SHRT_MAX, otherwise undefined</td>
</tr>
<tr>
<td>char c = CHAR_MAX; c++ -INT_MIN</td>
<td>varies(^1)</td>
</tr>
<tr>
<td>(char)INT_MAX</td>
<td>undefined(^2)</td>
</tr>
<tr>
<td>1&lt;&lt;-1</td>
<td>commonly -1</td>
</tr>
<tr>
<td>1&lt;&lt;0</td>
<td>undefined</td>
</tr>
<tr>
<td>1&lt;&lt;31</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>commonly INT_MIN in ANSI C and C++98; undefined in C99 and C++11(^2,3)</td>
</tr>
<tr>
<td>1&lt;&lt;32</td>
<td>undefined(^3)</td>
</tr>
<tr>
<td>1/0</td>
<td>undefined</td>
</tr>
<tr>
<td>INT_MIN%=-1</td>
<td>undefined in C11, otherwise undefined in practice</td>
</tr>
</tbody>
</table>

---

1. The question is: Does `c` get “promoted” to `int` before being incremented? If so, the behavior is well-defined. We found disagreement between compiler vendors’ implementations of this construct.
2. Assuming that the `int` type uses a two’s complement representation
3. Assuming that the `int` type is 32 bits long
Undefined Behavior in C/C++

- Undefined Behavior means anything can happen
  - Code may work as expected, return an error, or
  - Compiler may ignore undefined behavior code, or
  - GCC 1.17 would run nethack!

- In C and C++, signed overflows undefined
  - INT_MAX+1 may not wrap to INT_MIN

- Undefined behavior allows optimizations
  - If signed overflows undefined, compiler can assume
    X+1 > X is TRUE.
  - Allows compiler to assume (for i=0; i<=N, i++) executes exactly N+1 times, which permits many loop optimizations.
Undefined Behavior in Kernel

Function

```c
void agnx_pci_remove (struct pci_dev *pdev) {
    struct ieee80211_hw *dev = pci_get_drvdata(pdev);
    struct agnx_priv *priv = dev->priv;
    if (!dev) return;
    ... do stuff using dev ... 
}
```

Compiler

Case 1: dev == NULL
- dev->priv undefined, so compiler can do anything.

Case 2: dev != NULL
- NULL pointer check won’t fail, so optimize it out.

Result
- Compiler removes NULL pointer check from code.
Google NaCl Safety Check

Google Native Client (NaCl) for web applications.

During a routine refactoring, code that once read

```c
aligned_tramp_ret = tramp_ret & ~(nap->align_boundary - 1);
```

was changed to read

```c
return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);
```

introducing a bit shift.

Vulnerability

- NaCl on x86 uses 32-bit size, so nap->align_boundary was 32.
- `1<<32` is undefined in the C standard.
- Compiler replaced safety check with NOP.
Undefined Behavior Sanitizer

- Adds runtime checks to compiled programs for certain types of undefined behavior.
- GCC 4.9 added `-fsanitize=undefined` flag
  - `-fsanitize=shift`: error on undefined shifts.
  - `-fsanitize=integer-divide-by-zero`
  - `-fsanitize=integer-divide-by-zero`
  - `-fsanitize>null`: error message on NULL deref.
  - `-fsanitize=signed-integer-overflow`
Broward County 2004 election

- Amendment 4 vote was reported as tied.
- Software from ES&S Systems reported a large negative number of votes.
- Discovery revealed that Amendment 4 had passed by a margin of over 60,000 votes.
Integer overflows in Amarok media player.

- Reads input size + input from file.
- Allocates input size + 1 bytes, which can be very small if an overflow occurs.
- Reads file data into very small buffer, leading to a buffer overflow.
Integer Multiplication Overflow

CESA-2004-001: libpng

    info_ptr->row_pointers =
        (png_bytepp)png_malloc(png_ptr,
        info_ptr->height * sizeof(png_bytep));

If height > INT_MAX / sizeof(png_bytep)
Size of new buffer will be a small integer.
User data in image file can overflow buffer.
Widening Conversions

A conversion from a type with a smaller range of values to type with a larger range of values.

Examples: byte -> short, short -> long

Sign extension

Propagates signed bit from source type to all unused bits in destination type.

Magnitude and sign are preserved.
Widening Conversion Example

Source type: byte
Value: -7
1 1 1 1 1 0 0 1

Destination type: short
Value: -7
1 1 1 1 1 1 1 1 1 1 1 1 0 0 1
Narrowing Conversions

Conversions from a wider type to a narrower type.
Examples: long -> byte, int -> short

Truncation

Bits from source type that don’t fit into narrower destination type are discarded.
Magnitude and sign may change.
Narrowing Conversion Example

Source Type: short
Value: 257
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1

Destination Type: byte
Value: 1
0 0 0 0 0 0 0 0 1
Sign Extension Vulnerability

CERT CA-1996-22: bash

`yy_string_get()` reads user data as chars. Each char converted to an int when parsed. A char value of 255 sign extended to int -1. The integer -1 means command separator.

Example exploit

`bash -c 'ls\377who'`
Ariane 5 Crash

- Converted a 64-bit float into a 16-bit int, causing a software exception in both main and backup computers.
- Both computers crashed, resulting in loss of control.
Integer Mitigation Strategies

1. Appropriate integer type selection
2. Arbitrary-precision arithmetic
3. Precondition and postcondition checking
4. Range checking
5. Compiler checks
6. Secure integer libraries
Unsigned Addition Checks

An unsigned addition

```c
unsigned int x, y, sum;
sum = x + y;
```

Precondition

```c
if( x > UINT_MAX - y) /* error */
```

Postcondition

```c
if((x >= 0 && sum < y) || (x < 0 && sum > y)) /* error */
```
## Signed Addition Preconditions

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>Positive</td>
<td>if ( x &gt; \text{INT_MAX} - y ) /* error */</td>
</tr>
<tr>
<td>Positive</td>
<td>Negative</td>
<td>None</td>
</tr>
<tr>
<td>Negative</td>
<td>Positive</td>
<td>None</td>
</tr>
<tr>
<td>Negative</td>
<td>Negative</td>
<td>if ( x &lt; \text{INT_MIN} - y ) /* error */</td>
</tr>
</tbody>
</table>
Range Checking

Check that integer ranges are valid.

Be more specific than INT_MIN, INT_MAX.

Liquid water temperatures range 0..100 @ STP.

Use type system to check.

Some languages allow integer ranges.

Create abstract data types in languages that don’t provide integer range types.
Compiler Checks

Microsoft VS 2005 CL
- Runtime integer error checks: /RTCc
- Use highest warning level /W4
- Check for #pragma warning(disable, C####)

GCC
- Make signed integer overflow wrap: -fwrapv
- Runtime integer error checks: -ftrapv
- Use integer-relevant warnings: -Wconversion –Wsign-compare
- Check for #pragma GCC diagnostic ignored
Secure Integer Libraries

**IntegerLib**
- Designed for C, but usable in C++.
- Available from CERT.

**IntSafe**
- C library written by Michael Howard.
- Uses architecture specific inline assembly.

**SafeInt**
- C++ template class from David LeBlanc.
int main(int argc, char *const *argv) {
    try {
        SafeInt<unsigned long> s1(strlen(argv[1]));
        SafeInt<unsigned long> s2(strlen(argv[2]));
        char *buff = (char *) malloc(s1 + s2 + 1);
        strcpy(buff, argv[1]);
        strcat(buff, argv[2]);
    } catch(SafeIntException err) {
        abort();
    }
}
Key Points

1. Understand how integer arithmetic works.
   1. Two’s complement signed integers.
   2. Overflow handling: wrap, saturate, flag, exception, conversion to higher precision.
2. Undefined behavior in C/C++
   1. Compiler can do anything, often introducing vulnerabilities.
3. Security impacts of integer overflows
   1. Defeating bounds checks.
   2. Influence important counts, like votes.
   3. Setting special values.
4. Integer overflow mitigation strategies
   1. Preconditions and postconditions
   2. Range checking
   3. Compiler checking
   4. Secure integer libraries
References