Overview

- Integer errors are the source of a growing number of vulnerabilities, ones that have been paid little attention compared with buffer overflows, format string attacks etc.
- To understand integer vulnerabilities we first take a look at the various C integer types before going on to examine how they are represented inside your computer
- Along the way we look at the vulnerabilities that can stem from a lack of understanding of integer types and arithmetic operation side-effects
- Material drawn from...
    - *Secure Coding in C and C++* by Seacord
    - *The C Programming Language* by Kernighan and Ritchie

C Data Types and Sizes

Type	Description
`char`	A single byte
`int`	Integer, reflects the host's natural integer size
`float`	Single-precision floating point
`double`	Double-precision floating point

- Two additional qualifiers can be applied to integers, namely `long` and `short`
- These should provide different length integer types where practical, usually `short` is 16 bits, `long` 32 bits with `int` reflecting the host's natural size (typically 32 bits)

C Data Types and Sizes

- Two additional qualifiers exist, `signed` and `unsigned` and can be applied to a `char` or any integer type
- Unsigned numbers are always positive or zero and obey the laws of arithmetic modulo 2^n where `n` is the number of bits in the type e.g. a `char` has 8 bits so an `unsigned char` occupies the range `[0, 255]`
- With signed numbers one bit is given over to represent $\pm$ and thus a `signed char` occupies `[-128, 127]`
- Whether a plain `char` is `signed` or `unsigned` is machine dependent so be explicit if unsure
- But how are integers represented in the machine?

Unsigned Integers

- For `unsigned` integers the representation is straightforward and as we might expect, e.g. $00000011_2 = 3_{10}$ and $10001001_2 = 137_{10}$
- The range of an `unsigned` type is $[0, 2^n - 1]$ where `n` is the number of bits assigned to that type
- That's fine for `unsigned` types but how are we going to represent `signed` types?
- Ideally we'd like to represent them in such a way that subtraction requires no extra circuitry (i.e. existing addition circuitry works for subtraction)

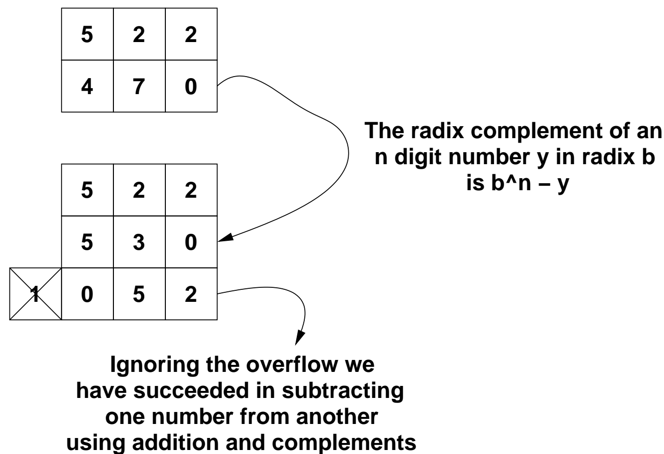
Unsigned Integers

- We'd also like to have a single representation for zero
- However, with a simple sign/magnitude approach we end up with two zeros, a positive zero and a negative zero
- Not a satisfactory solution so what can we do?

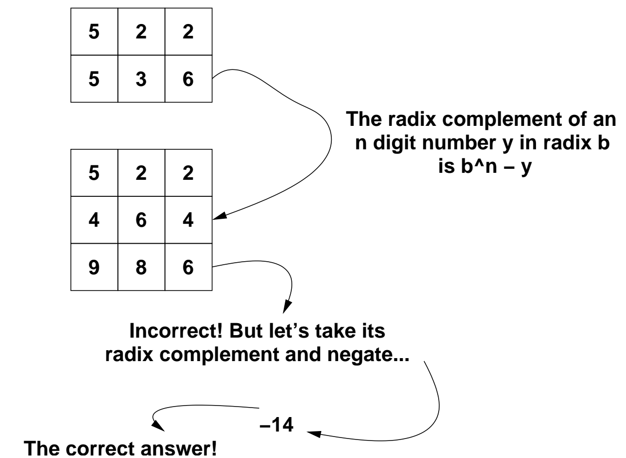
Complements

- One of our goals is to implement subtraction "through addition"
- The method of complements achieves this goal
- The approach is best illustrated with a couple of examples of $x - y$, firstly with $y < x$ and then with $y > x$
- For "radix", read "base"

Complements



Complements



Complements

- Mmm, we may be onto something here. . .
- If we represent negative numbers using their complements then subtraction can be implemented using addition circuitry and we'll get the correct answer for the last two examples
- In a sense taking the complement is both:
  - A means of representing negative numbers and
  - The operation of negation
- In the decimal examples so far we calculated the ten's complement of the number to be subtracted: this is the same as the nines' complement plus one

Complements

- On a computer we work with binary so rather than the ten's complement we've seen so far we'll use two's complement to represent negative numbers
- Just as ten's complement is nines' complement plus one, two's complement is ones' complement plus one: this turns out to be very handy
- Ones' complement is really easy to calculate: it's merely a case of applying a bitwise NOT operation to the binary representation of the corresponding positive number
- The following examples should make things clearer. . .

Complements

5	0	1	0	1	
	1	0	1	0	Ones' complement
	0	0	0	1	Plus 1
–5	1	0	1	1	Two's complement

6	0	1	1	0	
–5	1	0	1	1	–2^3 + 2^1 + 2^0
1	0	0	0	1	

Complements

7	0	1	1	1	Largest possible positive number
	1	0	0	0	Ones' complement
	0	0	0	1	Plus 1
–7	1	0	0	1	Two's complement

Largest possible negative number? No

–8	1	0	0	0	
	0	1	1	1	Ones' complement
	0	0	0	1	Plus 1
–8	1	0	0	0	Two's complement

Range: [–2^n, 2^n–1]

Complements

- What about zero? Well, as a `char` it's represented as `00000000` and taking its two's complement gives the same number (try it) so we have a single zero representation which was one of our goals
- We have lost one positive number to zero (it's counted as positive) and that's why ranges are not symmetric e.g. `[-128, 127]` for a `signed char`
- The leading bit is the sign bit, 0 for positive numbers and 1 for negative numbers
- For unsigned types this is the most significant bit

Complements

- Interpreted as unsigned, negatives are big. . .
- So if you disassemble a function in `gdb` and look at the code that makes space on the stack for local variables you'll see some big, ugly numbers beginning `0xF` (`gdb` is interpreting small signed as big unsigned numbers)
- Knowing how they are represented we now move on to examine what happens when we start using integers in arithmetic expressions and how integral promotion and arithmetic conversions can make things a little tricky

Arithmetic Expressions

- Consider the following C excerpt:

```
char a = 127;
char b = 1;
char c = -25;
short int x;
x = a + b + c;
```

- What value will `x` have?
- What stops the intermediate result of `(a + b)` overflowing the `signed char` range?

Arithmetic Expressions

Integral promotion

A character, a short integer, or an integer bit–field, all either signed or not, or an object of enumeration type, may be used in an expression wherever an integer may be used. If an int can represent all the values of the original type, then the value is converted to int; otherwise the value is converted to unsigned int. This process is called integral promotion.

"The C Programming Language", 2nd edition, Kernaghan & Ritchie, Appendix A6.1.

Arithmetic Expressions

- So in the last example we do not get an overflow because the intermediate value is held as an `int`
- What happens here?

```
unsigned char a = 5;
short b = 5;
float c = 5;
double x = a + b + c;
```

- How are mixed types handled in arithmetic expressions? We can't add apples and oranges
- Your C compiler applies the following set of rules. . .

Arithmetic Expressions

Arithmetic conversions

Arithmetic operands must be brought to a common type, which is also the type of the result. These are the usual arithmetic conversions:

1. If either operand is long double, the other is converted to long double.
2. Otherwise, if either operand is double, the other is converted to double.
3. Otherwise, if either operand is float, the other is converted to float.
4. Otherwise, the integral promotions are performed on both operands; then, if either operand is unsigned long int, the other is converted to unsigned long int.
5. Otherwise if one operand is long int and the other is unsigned int, the effect depends on whether a long int can represent all the values of an unsigned int; if so, the unsigned int operand is converted to long int; if not, both are converted to unsigned long int.
6. Otherwise, if one operand is long int, the other is converted to long int.
7. Otherwise, if either operand is unsigned int, the other is converted to unsigned int.
8. Otherwise, both operands have type int.

"The C Programming Language", 2nd edition, Kernaghan & Ritchie, Appendix A6.5.

Arithmetic Expressions

From	To	Method	?
`unsigned char`	`char`	Keep bit pattern	
`unsigned char`	`short`	Zero extend	
`unsigned char`	`long`	Zero extend	
`unsigned char`	`unsigned short`	Zero extend	
`unsigned char`	`unsigned long`	Zero extend	
`unsigned short`	`char`	Keep low byte	
`unsigned short`	`short`	Keep bit pattern	

Arithmetic Expressions

From	To	Method	?
`unsigned short`	`long`	Zero extend	
`unsigned short`	`unsigned char`	Keep low byte	
`unsigned long`	`char`	Keep low byte	
`unsigned long`	`short`	Keep low word	
`unsigned long`	`long`	Keep bit pattern	
`unsigned long`	`unsigned char`	Keep low byte	
`unsigned long`	`unsigned short`	Keep low word	

Arithmetic Expressions

From	To	Method	?
char	short	Sign extend	
char	long	Sign extend	
char	unsigned char	Keep bit pattern	
char	unsigned short	Sign extend to short; convert short to unsigned short	
char	unsigned long	Sign extend to long; convert long to unsigned long	
short	char	Keep low byte	
short	long	Sign extend	

Arithmetic Expressions

From	To	Method	?
short	unsigned char	Keep low byte	
short	unsigned short	Keep bit pattern	
short	unsigned long	Sign extend to long; convert long to unsigned long	
long	char	Keep low byte	
long	short	Keep low word	
long	unsigned char	Keep low byte	
long	unsigned short	Keep low word	
long	unsigned long	Keep bit pattern	

Integer Error Conditions

- Integer operations can give unexpected results in the presence of
  - Integer overflow
  - Sign errors
  - Truncation errors

Integer Overflow

- Occurs when an integer is increased beyond its maximum value or decreased below its minimum value (underflow)
- Overflows can be unsigned, for example,

```
unsigned char a = 255;
a++;
```

- Overflows can also be signed, for example,

```
signed char a = -128;
a--;
```

Integer Overflow Vulnerabilities

- Here is some “real world” code to handle comments in JPEG files that can lead to heap overflow vulnerabilities. . .

```
void getComment(unsigned int len, char *src) {
 unsigned int size;
 size = len - 2;
 char *comment = (char *) malloc(size + 1);
 memcpy(comment, src, size);
}
```

Sign Errors

- These errors occur when converting between signed and unsigned types
- When converting from signed to larger unsigned integer, the smaller is first sign extended; the high bit loses its negativity and we end up with a much bigger result than intended (if original value was negative)
- If signed integer was originally positive, no error occurs

Sign Errors

- Now apply the rules in the following example:

```
char a = -1;
unsigned int b = 1;
if (a > b) {
 printf(``-1 is greater than 1'');
}
```

- What is printed out? Why?
- What if we changed `b` to an `unsigned short`?

Sign Error Vulnerabilities

- A contrived example but illustrates the problem of “hidden” integer conversions leading to problems. . .

```
#define SIZE 10
int main(int argc, char *argv[]) {
 int len; char buffer[SIZE];
 len = atoi(argv[1]);
 if (len < SIZE) {
  memcpy(buffer, argv[2], len);
}
```

Truncation Errors

- Truncation errors occur when an integer is converted to a smaller type and its original value is outside the range of the new type, for example,

```
int a = 258;
unsigned char b = a;
```

- A change of sign may also occur, for example,

```
int a = 255;
char b = a;
```

- May also occur when integers are of the same size

Truncation Error Vulnerabilities

- Here the author has tried to be careful but is undone by integer truncation effects...

```
int main(int argc, char *argv[]) {
 unsigned short int total;
 total = strlen(argv[1]) + strlen(argv[2]) + 1;
 char *buffer = (char *)malloc(total);
 strcpy(buffer, argv[1]);
 strcat(buffer, argv[2]);
 return (0);
}
```

Mitigation Strategies

- Range checking
- Strong typing
- Compiler-generated runtime checks
- Libraries and classes for integer handling
  - Howard's C library
  - LeBlanc's SafeInt class
  - The RCSint class

Detecting/Preventing Exceptional Conditions

- Preconditions can be used to detect that an error will occur before carrying out an operation
- Error detection requires determining if an error occurred while performing an operation
- The postcondition technique performs the operation and tests the resulting value for validity
- Addition
  - Overflow conditions can be detected on IA-32
  - A carry flag signals unsigned overflow
  - An overflow flag signals signed overflow

Detecting/Preventing Exceptional Conditions

- Thus, `jc` and `jo` assembly instructions can be used to detect if an error occurred during the addition
- Preconditions can be constructed: for no overflow to occur it is necessary that operands sum to a value that fits in the destination's range (just make sure intermediate values don't overflow)
- Postconditions may also be tested: if we add two `unsigned ints`, `a` and `b`, the result should be bigger than `a` and `b`; if not, an overflow occurred
- Similar tests can be carried out for subtraction, division and multiplication

Range Checking

- Sometimes it's easy to apply range checking e.g. when indexing into an array
- On other occasions it may not be so easy e.g. `x = a * b * c` where `a`, `b`, `c` themselves are the result of previous calculations
- Making types unsigned where negative values are nonsensical may help e.g. on array indices
- One approach is to identify all external integer inputs to the program, attempt to determine valid ranges for them and apply range checking at the interface

Strong Typing

- Create a C++ class where data is marked private and public get/set methods are used to apply range checking
- In C an enumeration variable might (depends on the compiler) prove useful for limiting ranges to reasonable values e.g.
  `enum weekDays  {Mon, Tue, Wed, Thu, Fri};`

Compiler-Generated Runtime Checks

- Can ask `gcc` to generate runtime traps for signed overflow on addition, subtraction and multiplication by specifying the `-ftrapv` flag
- Calls are automatically made to functions that test postconditions and send a `SIGABRT` to the process on error

Libraries and Classes

C library

- Written by Michael Howard and detects overflow through checking IA-32 flags
- To add two integers we call the appropriate library function and pass it the integers as arguments
- Efficient but not portable across architectures

Libraries and Classes

SafeInt class

- A C++ template class by David LeBlanc that implements checking through preconditions
- Integers are created as SafeInt objects
- Portable and easier to use through operator overloading but slower than Howard's approach since it does not use any architecture specific flags

Libraries and Classes

RCSint class

- “Reliable, Convenient and Secure Integers”
- C++ template class with assembly coding for speedup
- Easy to use at the expense of portability again