IS 2620

Secure Coding in C and C++

Integer Security

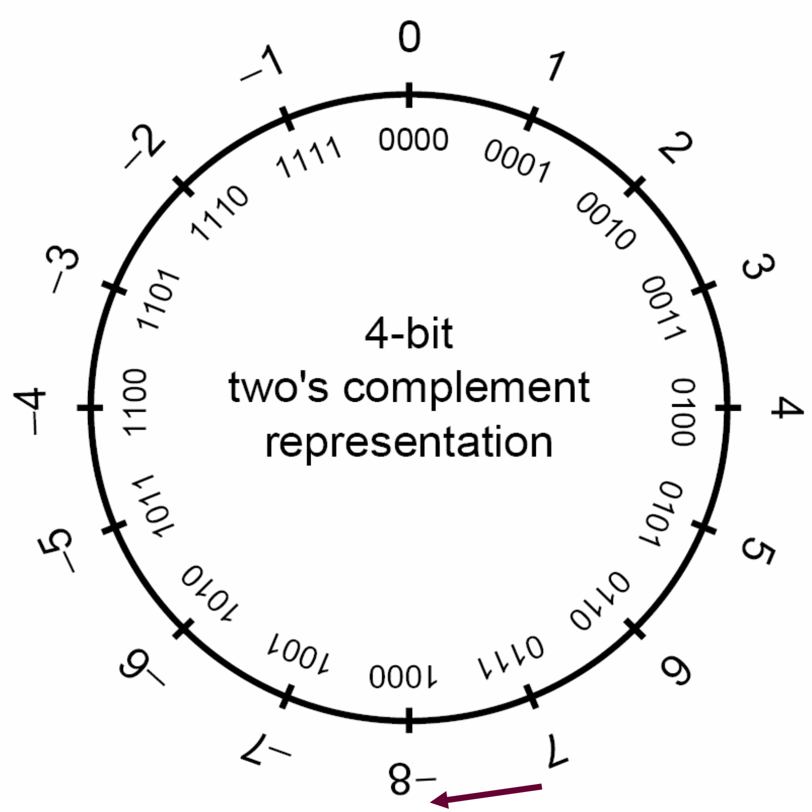
Lecture 7

Acknowledgement: These slides are based on author Seacord’s original presentation

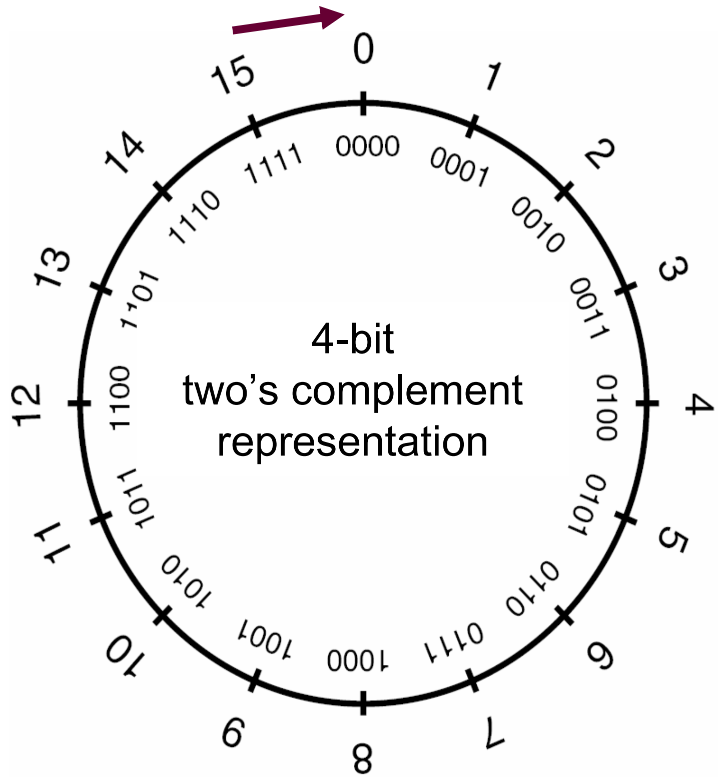
Integer Security

- Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs.
- Integer range checking has not been systematically applied in the development of most C and C++ software.
  - security flaws involving integers exist
  - a portion of these are likely to be vulnerabilities
- A software vulnerability may result when a program evaluates an integer to an unexpected value.

Representation

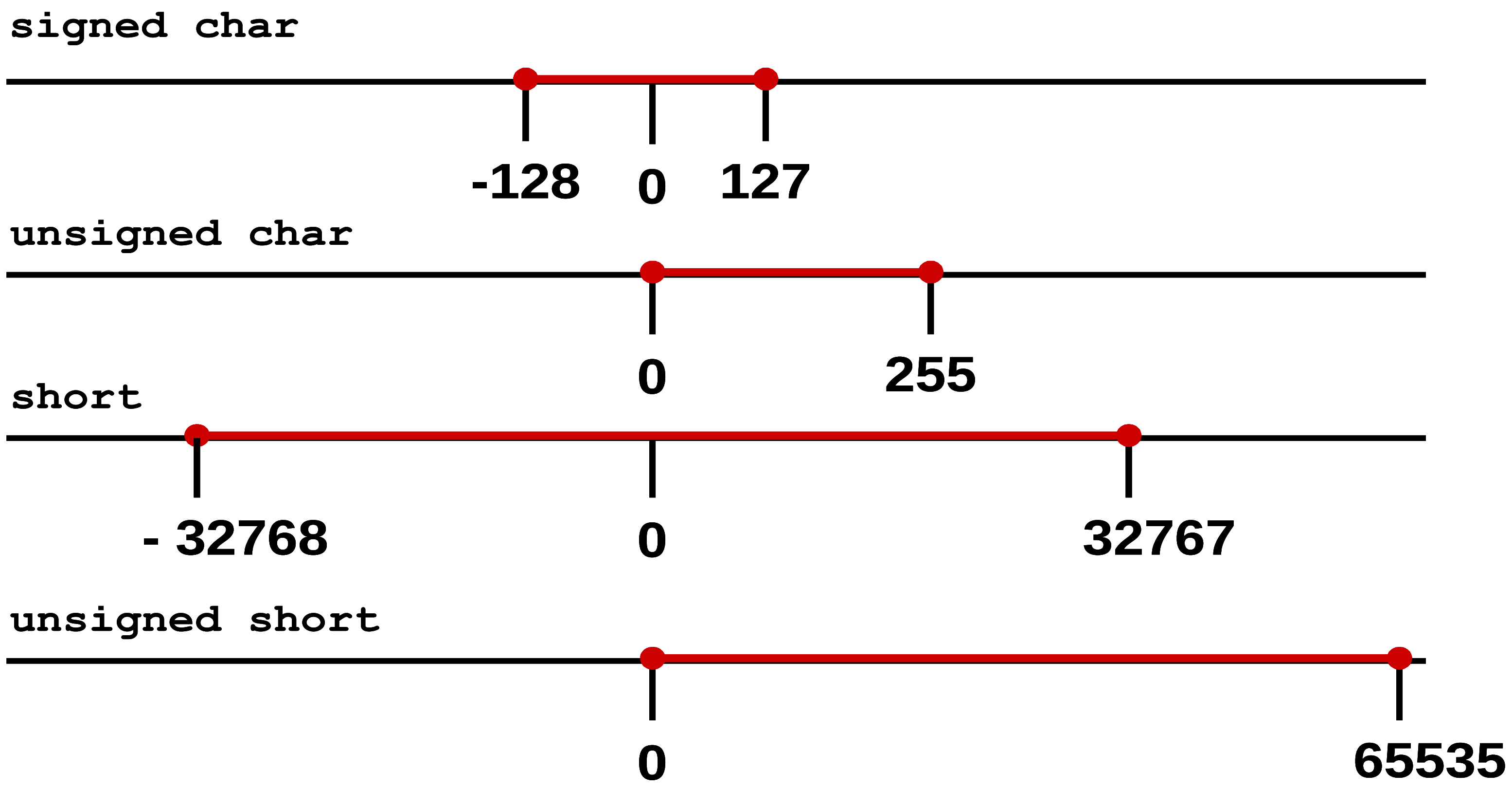


Signed Integer



Unsigned Integer

Example Integer Ranges



Integer Promotion Example

- Integer promotions require the promotion of each variable (**`c1`** and **`c2`**) to **`int`** size

```
char c1, c2;
c1 = c1 + c2;
```

  - The two **`ints`** are added and the sum truncated to fit into the **`char`** type.
  - Integer promotions avoid arithmetic errors from the overflow of intermediate values.

Implicit Conversions

The sum of **c1** and **c2** exceeds the maximum size of **signed char**

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. c3 = -120;
5. cresult = c1 + c2 + c3;
```

However, **c1**, c1, and **c3** are each converted to integers and the overall expression is successfully evaluated.

The sum is truncated and stored in **cresult** without a loss of data

The value of **c1** is added to the value of **c2**.

From unsigned	To	Method
char	char	Preserve bit pattern; high-order bit becomes sign bit
char	short	Zero-extend
char	long	Zero-extend
char	unsigned short	Zero-extend
char	unsigned long	Zero-extend
short	char	Preserve low-order byte
short	short	Preserve bit pattern; high-order bit becomes sign bit
short	long	Zero-extend
short	unsigned char	Preserve low-order byte
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	long	Preserve bit pattern; high-order bit becomes sign bit
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word

Key: Lost data | Misinterpreted data

From	To	Method
char	**short**	**Sign-extend**
char	**long**	**Sign-extend**
char	**unsigned char**	**Preserve pattern; high-order bit loses function as sign bit**
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**	**Preserve low-order byte**
short	**long**	**Sign-extend**
short	**unsigned char**	**Preserve low-order byte**
short	**unsigned short**	**Preserve bit pattern; high-order bit loses function as sign bit**
short	**unsigned long**	**Sign-extend to long; convert long to unsigned long**
long	**char**	**Preserve low-order byte**
long	**short**	**Preserve low-order word**
long	**unsigned char**	**Preserve low-order byte**
long	**unsigned short**	**Preserve low-order word**
long	**unsigned long**	**Preserve pattern; high-order bit loses function as sign bit**

Key: Lost data | Misinterpreted data

Signed Integer Conversion Example

- **`1. unsigned int l = ULONG_MAX;`**
- **`2. char c = -1;`**
- **`3. if (c == l) {`**
- **`4.  printf("-1 = 4,294,967,295?\n");`**
- **`5. }`**

The value of **`c`** is compared to the value of **`l`**.

Because of integer promotions, **`c`** is converted to an unsigned integer with a value of **`0xFFFFFFFF`** or 4,294,967,295

Overflow Examples 1

```
1. int i;
2. unsigned int j;

3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);

6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
```

i=-2,147,483,648

j = 0

Overflow Examples 2

- ` 9. i = INT_MIN; // -2,147,483,648;`
- `10. i--;`
- `11. printf("i = %d\n", i);`

`i=2,147,483,647`

- `12. j = 0;`
- `13. j--;`
- `14. printf("j = %u\n", j);`

`j = 4,294,967,295`

Truncation Error Example

- `1. char cresult, c1, c2, c3;`
- `2. c1 = 100;`
- `3. c2 = 90;`
- `4. cresult = c1 + c2;`

Adding **`c1`** and **`c2`** exceeds the max size of **`signed char (+127)`**

Truncation occurs when the value is assigned to a type that is too small to represent the resulting value

Integers smaller than **`int`** are promoted to **`int`** or **`unsigned int`** before being operated on

Sign Error Example

- `1. int i = -3;`
- `2. unsigned short u;`
- `3. u = i;`
- `4. printf("u = %hu\n", u);`

Implicit conversion to smaller unsigned integer

There are sufficient bits to represent the value so no truncation occurs. The two's complement representation is interpreted as a large signed value, however, so **`u = 65533`**

Integer Division

- An integer overflow condition occurs when the minimum integer value for 32-bit or 64-bit integers are divided by -1.
  - In the 32-bit case, –2,147,483,648/-1 should be equal to 2,147,483,648

- 2,147,483,648 /-1 = - 2,147,483,648

  - Because 2,147,483,648 cannot be represented as a signed 32-bit integer the resulting value is incorrect

Vulnerabilities Section Agenda

- Integer overflow
- Sign error
- **Truncation**
- Non-exceptional

JPEG Example

- Based on a real-world vulnerability in the handling of the comment field in JPEG files
- Comment field includes a two-byte length field indicating the length of the comment, including the two-byte length field.
- To determine the length of the comment string (for memory allocation), the function reads the value in the length field and subtracts two.
- The function then allocates the length of the comment plus one byte for the terminating null byte.

Integer Overflow Example

```
void getComment(unsigned int len, char *src) {
  unsigned int size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}
int _tmain(int argc, _TCHAR* argv[]) {
  getComment(1, "Comment ");
  return 0;
}
```

0 byte `malloc()` succeeds

Size is interpreted as a large positive value of `0xffffffff`

Possible to cause an overflow by creating an image with a comment length field of 1

Sign Error Example 1

Program accepts two arguments (the length of data to copy and the actual data)

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

`len` declared as a signed integer

`argv[1]` can be a negative value

A negative value bypasses the check

Value is interpreted as an unsigned value of type `size_t`

Sign Errors Example 2

- The negative length is interpreted as a large, positive integer with the resulting buffer overflow
- This vulnerability can be prevented by restricting the integer `len` to a valid value
  - more effective range check that guarantees `len` is greater than 0 but less than `BUFF_SIZE`
  - declare as an unsigned integer
    - eliminates the conversion from a signed to unsigned type in the call to `memcpy()`
    - prevents the sign error from occurring

Truncation: Vulnerable Implementation

```
bool func(char *name, long cbBuf) {
  unsigned short bufSize = cbBuf;
  char *buf = (char *)malloc(bufSize);
  if (buf) {
    memcpy(buf, name, cbBuf);
    if (buf) free(buf);
    return true;
  }
  return false;
}
```

`cbBuf` is used to initialize **`bufSize`** which is used to allocate memory for **`buf`**

`cbBuf` is declared as a long and used as the size in the **`memcpy()`** operation

Vulnerability 1

- `cbBuf` is temporarily stored in the unsigned short `bufSize`.
- The maximum size of an **`unsigned short`** for both GCC and the Visual C++ compiler on IA-32 is 65,535.
- The maximum value for a **`signed long`** on the same platform is 2,147,483,647.
- A truncation error will occur on line 2 for any values of `cbBuf` between 65,535 and 2,147,483,647.

Vulnerability 2

- This would only be an error and not a vulnerability if **`bufSize`** were used for both the calls to **`malloc()`** and **`memcpy()`**
- Because **`bufSize`** is used to allocate the size of the buffer and **`cbBuf`** is used as the size on the call to **`memcpy()`** it is possible to overflow **`buf`** by anywhere from 1 to 2,147,418,112 (2,147,483,647 - 65,535) bytes.

Negative Indices

```
int *table = NULL;\
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

Storage for the array is allocated on the heap

`pos` is not > 99

`value` is inserted into the array at the specified position

Vulnerability

- There is a vulnerability resulting from incorrect range checking of `pos`
  - Because `pos` is declared as a signed integer, both positive and negative values can be passed to the function.
  - An out-of-range positive value would be caught but a negative value would not.

Mitigation

- Type range checking
- Strong typing
- Compiler checks
- Safe integer operations
- Testing and reviews

Type Range Checking Example

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
    unsigned int len;
    char buf[BUFF_SIZE];
    len = atoi(argv[1]);
    if ((0<len) && (len<BUFF_SIZE) ){
        memcpy(buf, argv[2], len);
    }
    else
        printf("Too much data\n");
}
```

Implicit type check from the declaration as an unsigned integer

Explicit check for both upper and lower bounds

Strong Typing

- One way to provide better type checking is to provide better types.
- Using an unsigned type can guarantee that a variable does not contain a negative value.
- This solution does not prevent overflow.
- Strong typing should be used so that the compiler can be more effective in identifying range problems.

Strong Typing Example

- Declare an integer to store the temperature of water using the Fahrenheit scale
  - `unsigned char waterTemperature;`
- `waterTemperature` is an unsigned 8-bit value in the range 1-255
- `unsigned char`
  - sufficient to represent liquid water temperatures which range from 32 degrees Fahrenheit (freezing) to 212 degrees Fahrenheit (the boiling point).
  - does not prevent overflow
  - allows invalid values (e.g., 1-31 and 213-255).

Abstract Data Type

- One solution is to create an abstract data type in which `waterTemperature` is private and cannot be directly accessed by the user.
- A user of this data abstraction can only access, update, or operate on this value through public method calls.
- These methods must provide type safety by ensuring that the value of the `waterTemperature` does not leave the valid range.
- If implemented properly, there is no possibility of an integer type range error occurring.

Safe Integer Operations 1

- Integer operations can result in error conditions and possible lost data.
- The first line of defense against integer vulnerabilities should be range checking
  - Explicitly
  - Implicitly - through strong typing
- It is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program.

Safe Integer Operations 2

- An alternative or ancillary approach is to protect each operation.
- This approach can be labor intensive and expensive to perform.
- Use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source.

SafeInt Class

- SafeInt is a C++ template class written by David LeBlanc.
- Implements a precondition approach that tests the values of operands before performing an operation to determine if an error will occur.
- The class is declared as a template, so it can be used with any integer type.
- Every operator has been overridden except for the subscript **`operator[]`**

Testing 1

- Input validation does not guarantee that subsequent operations on integers will not result in an overflow or other error condition.
- Testing does not provide any guarantees either
  - It is impossible to cover all ranges of possible inputs on anything but the most trivial programs.
  - If applied correctly, testing can increase confidence that the code is secure.

Testing 2

- Integer vulnerability tests should include boundary conditions for all integer variables.
  - If type range checks are inserted in the code, test that they function correctly for upper and lower bounds.
  - If boundary tests have not been included, test for minimum and maximum integer values for the various integer sizes used.
- Use white box testing to determine the types of integer variables.
- If source code is not available, run tests with the various maximum and minimum values for each type.

Source Code Audit

- Source code should be audited or inspected for possible integer range errors
- When auditing, check for the following:
  - Integer type ranges are properly checked.
  - Input values are restricted to a valid range based on their intended use.
- Integers that do not require negative values are declared as unsigned and properly range-checked for upper and lower bounds.
- Operations on integers originating from untrusted sources are performed using a safe integer library.

Notable Vulnerabilities

- Integer Overflow In XDR Library
  - SunRPC xdr_array buffer overflow
  - http://www.iss.net/security_center/static/9170.php
- Windows DirectX MIDI Library
  - eEye Digital Security advisory AD20030723
  - http://www.eeye.com/html/Research/Advisories/AD20030723.html
- Bash
  - CERT Advisory CA-1996-22
  - http://www.cert.org/advisories/CA-1996-22.html