FOUNDATIONS OF CYBERSECURITY

C and C++ Secure Coding

Gianluca Dini
Dept. of Information Engineering
University of Pisa
Email: gianluca.dini@unipi.it

Version: 2021-03-03

1

Credits

UNIVERSITÀ DI PISA

- These slides come from a version originally produced by Dr. Pericle Perazzo

C and C++ Secure Coding

DYNAMIC MEMORY MANAGEMENT

3

Invalid Pointers

UNIVERSITÀ DI PISA

TAINTED

```
char* func(size_t size, size_t index) {
  if (index >= size) {
    return NULL;
  }
  char *buffer = (char*)malloc(size);
  buffer[index] = 'A';
  return buffer;
}
```

CONTROLLIAMO CHE NON SUCCEDE.

SE LA MALLOC FALLISCE RITORNA NULL = Ø

VA CONTROLLATO

*(buffer + index)

A QUESTO PUNTO L'AVVERSARIO PUO' ACCEDERE AD OGNI VALORE INDEX.

This function allocates a block of dynamic memory of "size" bytes in the heap, and then writes some data on position "index". The values of both the "size" and the "index" arguments are tainted.
The function does not check if the allocation succeeds, that is, if the malloc() method returns NULL. An attacker can provoke this by sending a properly large "size" input. The NULL pointer is then added to "index", thus forming an arbitrary valid pointer, over which data is written.

Invalid Pointers

UNIVERSITÀ DI PISA

```
char* func(size_t size, size_t index) {
  if (index >= size) {
    return NULL;
  }
  char *buffer = (char*)malloc(size);
  if (!buffer) { /* Handle error */ }
  buffer[index] = 'A';
  return buffer;
}
```

5

The malloc() function must always be checked for allocation failure.
Do not assume infinite heap space.
Do not assume that malloc initializes memory to all bits zero.

Invalid Pointers

std:: strings

- C++ strings are safer than C → ARRAY DI CARATTERI
  - Automatic allocation/deallocation
  - Managed string terminator
- C++ allocation is safer than C
  - new and new[] raise exceptions on failure
- C++ vectors are NOT MUCH safer than C arrays
  - Copy/range constructors allocate space
  - BUT: no bounds checking!

6

C++ is safer than C regarding memory allocation/deallocation. Indeed, the new and new[] operators (by default) raise an exception if the allocation fails, so it is harder to forget to handle such error than using malloc().
However, STL containers like std::vector and STL iterators are not much safer than "classic" C arrays, since they follow the same power-to-the-programmer philosophy, and they are efficiency-oriented. Some container contructors (namely the copy constructor and the range constructor) are useful, because they automatically allocate the right space. (Conversely, std::string is inherently much safer than "classic" C strings, since they realize complex functions like concatenations (operator+) by automatically managing allocation/deallocation/reallocation. In addition, they automatically manage the end-of-string terminator.)

STL → CONTAINERS

std::fill_n → RIEMPIE UN CONTAINER DA UNA CERTA POSIZIONE PER UNA CERTA QUANTITA' (v.begin, 10, 0x42)
STESSI PROBLEMI

std::copy()

TAINTED

```
void func(const std::vector<int> &src) {
  std::vector<int> dest;
  std::copy(src.begin(), src.end(), dest.begin());
}
```

1. NON FA BOUNDARY CHECK
2. NON FA MEMORY REALLOCATION
   SE IL VETTORE SRC E' PIU' GRANDE DEL DEST
   NON VIENE GESTITA IN MANIERA
   PREVEDIBILE.

7

This function makes a local copy of the input vector, using the standard function std::copy(). As usual, a reference to a std::vector is tainted when it references to a valid and well-formed std::vector, but the number and value of its elements are tainted.
The std::copy() function does not implement any bound checking, and does not expand the destination vector, so it can lead to buffer overflows. std::copy() should always be used if the destination container has enough pre-allocated space to contain all the source elements. Also, the function std::fill_n(v.begin(), 10, 0x42) is dangerous for the same reasons.

std::copy()

```
void func(const std::vector<int> &src) {
  std::vector<int> dest(src.size());
  std::copy(src.begin(), src.end(), dest.begin());
}
```

```
void func(const std::vector<int> &src) {
  std::vector<int> dest(src); // COSTRUTTORE DI COPIA
}
```

8

A solution is to preallocate enough elements in the destination container. However, it is still error-prone. A better solution is to use the copy constructor. If you have to copy only a range of elements you can use the range constructor, which takes a begin and end iterator: std::vector<int> dest(some_begin_iterator(), some_end_iterator()).

Iterator Arithmetic

UNIVERSITÀ DI PISA

il compilatore sceglie il tipo giusto dell'iteratore

```
void func(const std::vector<int> &c) {
  auto e = c.begin() + 20;
  for (auto i = c.begin(); i != e; i++) {
    std::cout << *i;
  }
}
```

```
void func(const std::vector<int> &c) {
  auto e = c.begin() + (c.size()<20?c.size():20);
  for (auto i = c.begin(); i != e; i++) {
    std::cout << *i;
  }
}
```

This function prints on screen the first 20 elements of the input vector “c”. (From C11 on, the “auto” keyword before a variable definition tells the compiler to automatically deduce the variable type from the return type of the initializer. It is useful to declare iterator, since their full type name is quite long. It also makes the code more maintainable, if you change the element type or the container type. Std::vector does not protect the programmer from the case in which the “c” vector has less than 20 elements, resulting in a out-of-bound read.

Out-of-Bound Access

UNIVERSITÀ DI PISA

```
std::vector<int> table(100);
int func(size_t index) {
  return table[index];
}
```

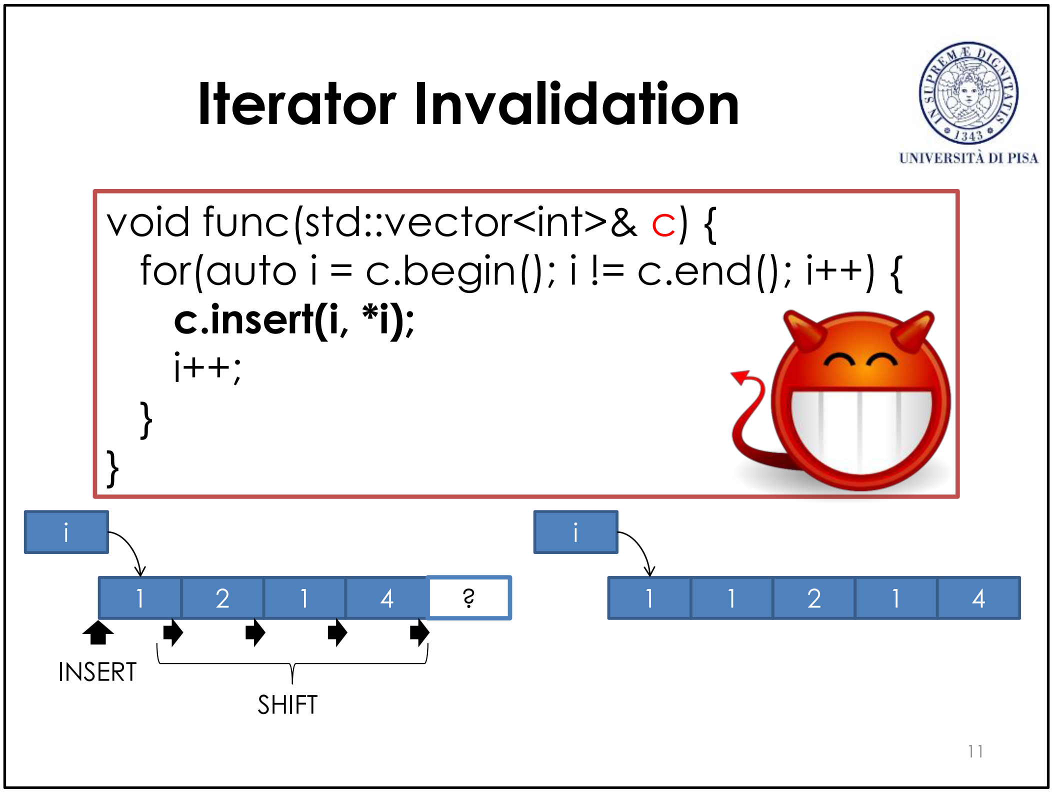
no boundary check

```
std::vector<int> table(100);
int func(size_t index) {
  return table.at(index);
}
```

out of range exception

10

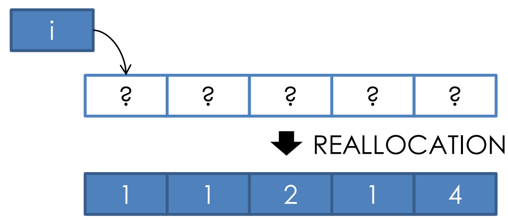
This function returns the value of the element of “table” at position “index”. However, the operator [] of std::vector does not check for bounds, resulting in an out-of-bound access. table.at(index) performs bound checking.



C++ iterators are very powerful as they permit the programmer to do things impossible with higher-language iterators, for example with Java iterators. One of these things is the on-the-fly modification of containers. This function takes an std::vector "c" and doubles every element by means of the method insert(). For example, the vector: 1 2 1 4 becomes: 1 1 2 2 1 1 4 4. The method insert(i, *i) inserts a new element o value "*i" in a container at the position specified by iterator "i" and shifts all the following elements towards right.

Iterator Invalidation

UNIVERSITÀ DI PISA



12

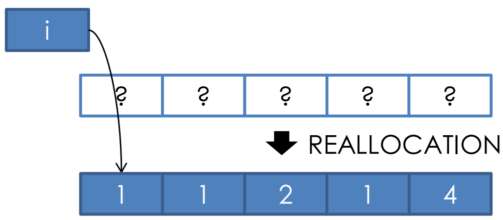
The insert() method could cause the reallocation of the std::vector internal buffer in case its capacity was not enough to accommodate the new element. In such a case, all the old iterators will be invalidated. The subsequent dereferencing of an invalid iterator constitutes an undefined behavior (typically, a read or write in unallocated memory).

Iterator Invalidation

UNIVERSITÀ DI PISA

```
void func(std::vector<int>& c) {
  for(auto i = c.begin(); i != c.end(); i++) {
    i = c.insert(i, *i);
    i++;
  }
}
```

RITORNA UN ITERATOR



13

In general, all the methods that invalidate iterators (e.g., insert(), erase(), push_back(), pop_back(), etc.) must be used carefully inside iterator-based cycles. The STL standard specifies which method of which container may invalidate which iterator. The standard solution is to use the return value of the insert() method, which always returns a valid iterator pointing to the newly inserted element. The erase() method return a valid iterator pointing to the element successive to the erased one.