Memory Safety

A mini-lecture series

CSE498 Collaborative Design (W) - Secure and Efficient C++ Software Development

02/11/2026

Kira Chan

https://cse.msu.edu/~chanken1/

Why is this an issue?

- C and C++ are low-level languages, which means they allow access and manipulation to memory directly as the developer
- To optimize for speed, C/C++ also does not do a lot of “safety” checks like other languages may
- This is a double-edged sword
  - Allows you to do some efficient and fast operations
  - But also requires you, as the developer, to be very cautious

Some stats

- Google reported that 70% of severe security bugs are actually memory issues [1]
- There is a recent push for memory-safe languages from the government (last several year)
  - Safecpp.org

What are some memory issues?

- Uninitialised memory
- Memory leaks
- Dangling Pointers
- Use after free
- Freeing multiple times
- Out of bounds memory, buffer overflow attacks, copying memory

Uninitialised Memory

Uninitialised Memory

```
1 int main(){
2     int x;
3     std::cout << (x) ;
4 }
5
```

- Which of these best describe the output of the code?
  a) Always defaults to 0
  b) Always std::numeric_limits<T>::min
  c) It is different every time
  d) The program will always crash
  e) Depends on the architecture of the compilation target, but always the same (platform dependent)

Uninitialised Memory

```cpp
int main(){
    int x;
    std::cout << (x) ;
}

```

- Which of these best describe the output of the code?
  a) Always defaults to 0
  b) Always std::numeric_limits<T>::min
  c) It is different every time
  d) The program will always crash
  e) Depends on the architecture of the compilation target, but always the same (platform dependent)

Uninitialised Memory

```
1 int main(){
2     int x;
3     std::cout << (x) ;
4 }
5
```

- What is the output of this code?
  - It is different every time
- This is undefined behaviour
  - Typically, x is going to be whatever bits are in the memory when the address is assigned to the program (treated as an int)

Uninitialized memory: class example

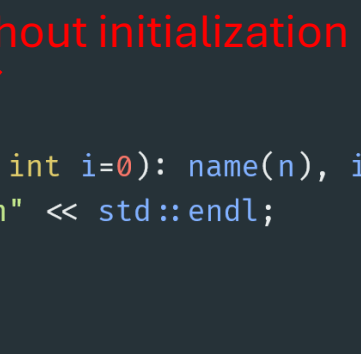
```cpp
class MyClass {
    private:
        std::string name;
        int id;
        int num;

    public:
        MyClass(std::string n="default", int i=0): name(n), id(i) {
            std::cout << "Object creation" << std::endl;
        }

        int getNum() {
            return num;
        }

};

int main(){
    MyClass myclass;
    int n = myclass.getNum();
}
```



Fixes

- Always initialize your variables before use
  - Unless you can only set it with information in constructor
- Default member initialisers (if applicable)

```
1  class MyClass {
2      private:
3          std::string name;
4          int id;
5          int num;
6
7          std::string name{};
8          int id{0};
9          int num{0};
10
```

Not initialised

Default member initialized (should always do this – see guideline video)

Memory Leaks

Memory leaks

- What is wrong with this code?

```
void foo() {
    int* p = new int(42);
}
```

- Challenges with new and passing the pointer around:
  - Who is supposed to free it?

Calculator memory leak?

- https://medium.com/quilltech/the-great-software-quality-collapse-how-we-normalized-catastrophe-96a6d51715f3

AI generated code is not free from exploits

- Expertise is need to be able to look and make sure AI does not generate code that has vulnerabilities / code issues

- From the article:
  - AI-generated code contains **322% more security vulnerabilities**
  - **45% of all AI-generated code** has exploitable flaws
  - Junior developers using AI cause damage **4x faster** than without it
  - **70% of hiring managers** trust AI output more than junior developer code

  - Source: https://medium.com/quilltech/the-great-software-quality-collapse-how-we-normalized-catastrophe-96a6d51715f3

Fixes

- Use Smart pointers
- Avoid owning raw pointers
- Adhere to Resource Acquisition is Initialisation (RAII)

Dangling pointers

Example 1: raw pointers

- What is a “dangling pointer”?

```
int main(){
    int* ptr = new int(20);
    delete ptr;
    ptr = nullptr;
    std::cout << *ptr;
}
```

  - std::cout << *ptr; // dereferencing the ptr leads to undefined behaviour

Example 1: Consequences

- Memory corruption
  - If the memory is reallocated elsewhere, then it can lead to memory corruption or overwrite the data
- Information leaks
  - Can lead to access to sensitive data if it is put there, or to privilege escalation (if that memory is used in security checks)
- Malicious code execution
  - If the pointer is used to make a virtual function call, then a different address (pointing to exploit code) can be called due to vtable pointer being overwritten

Use after free

- When a dangling pointer is used after it has been freed without allocating a chunk of memory, it is known as a “use after free” vulnerability
- CVE-2014-1776 [2]: In Microsoft Internet Explorer version 6-11, this vulnerability allowed attackers to execute arbitrary code or cause DoS attack using memory corruption
- Exploited in the wild!

Fixes

- **Use Smart pointers**
- Avoid owning raw pointers
- Adhere to Resource Acquisition is Initialisation (RAII)
- Be sure to always set the ptr back to nullptr

Freeing multiple times

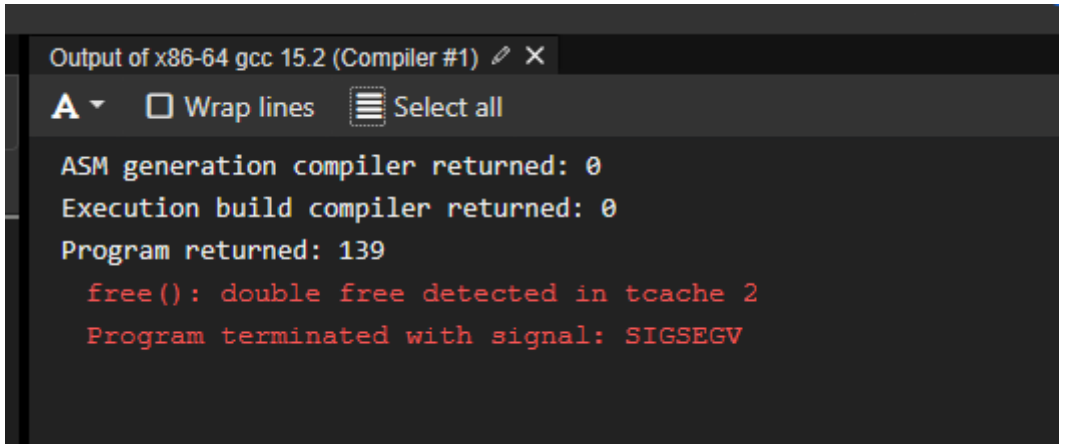
Deleting a ptr twice (or more)

- What happens here?
  a) Program immediate crashes (on line 9)
  b) Heap corruption (program crashes later)
  c) Potentially opens up to security vulnerability
  d) You can do this with certain types of pointers, but not others
  e) You are allowed to delete the same ptr twice in a row (compiler automatically will ignore the second delete as if it was not there)

```
1  int main(){
2      int* ptr = new int(20);
3      delete ptr;
4
5    /*
6    Much more code here in between
7    */
8
9      delete ptr;
10 }
```

Deleting a ptr twice (or more)

- What happens here?
  - a) **Program immediate crashes (on line 9)**
    - Modern memory allocators might actually detect this
    - Crashes your program immediately
    - GCC 15.2:



  - b) Heap corruption (program crashes later)
  - c) Potentially opens up to security vulnerability

Deleting a ptr twice (or more)

- What happens here?
  - a) **Program immediate crashes (on line 9)**
  - b) **Heap corruption (program crashes later)**
  - c) **Potentially opens up to security vulnerability**
  - d) You can do this with certain types of pointers, but not others
  - e) You are allowed to delete the same ptr twice in a row (compiler automatically will ignore the second delete as if it was not there)

```
1  int main(){
2      int* ptr = new int(20);
3      delete ptr;
4
5    /*
6    Much more code here in between
7    */
8
9      delete ptr;
10 }
```

Deleting a ptr twice (or more)

- What happens here?
  a) Program immediate crashes (on line 9)
  b) Heap corruption (program crashes later)
    - Bad outcome
    - Your program might use that memory later since it is freed
    - Second *delete ptr* is called
    - Corrupted Heap
    - Program crashes non-deterministically
  c) Potentially opens up to security vulnerability

Deleting a ptr twice (or more)

- What happens here?
  a) Program immediate crashes (on line 9)
  b) Heap corruption (program crashes later)
  c) Potentially opens up to security vulnerability
    - Attackers can influence heap structure
    - Later allocations can return an attacker-controlled memory / address
    - Function pointer or vtable pointers is maliciously overwritten
    - Arbitrary code exectuion

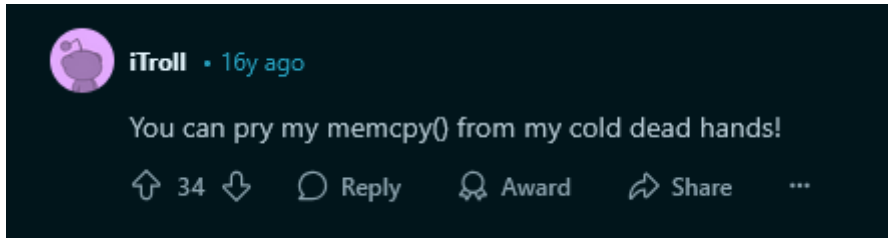
Fixes

- Don’t do it?
- Use smart pointers

Out of bounds memory, buffer overflow attacks, copying memory

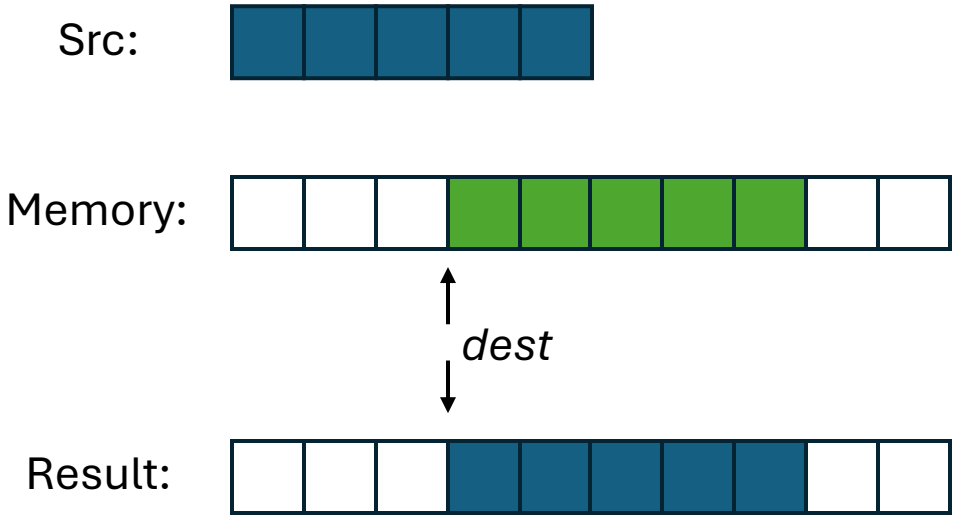
Copying memory

- The C native functions memcpy() and strcpy() are some of the most dangerous functions
- Yet, they are very popular in the C community
- In fact, Microsoft banned their use almost 20 years ago in 2009 in their secure development cycle
- You may be interacting or using code that include these functions



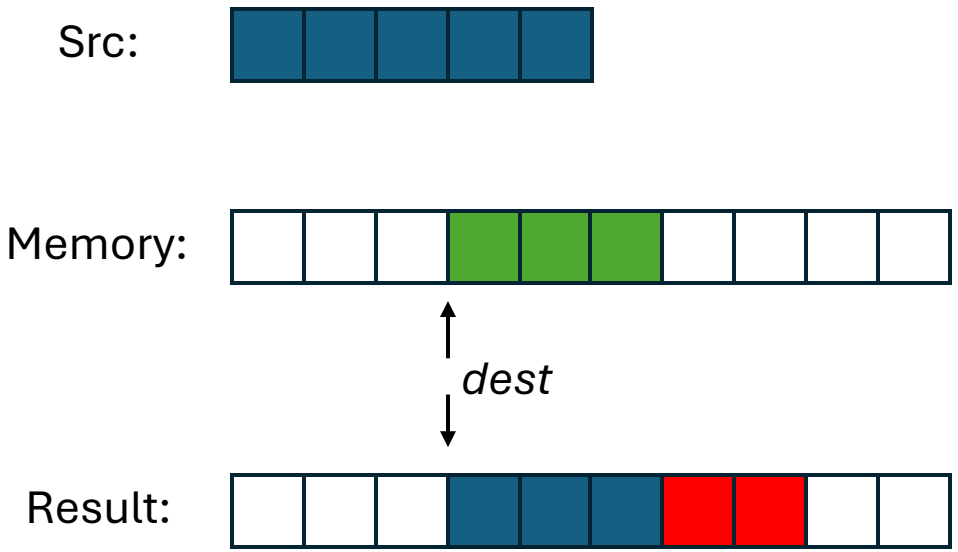
Memcpy() and strcpy()

- As the name suggest, it copies a buffer over from a *src* to a *dest*
- Assume *dest* is a buffer of char*[5]
- Ex: memcpy(*dest, src, 5)*



Buffer Overflow Attack

- If *dest* is a buffer of char*[3]
- Ex: memcpy(*dest, src, 5)*



- This becomes a really big problem if red contains code, or if sensitive information
- Imagine if red contains a bit that is True if user is authenticated

Underlying issue

- Memcpy() and strcpy() does not do bounds checking
  - It is assumed that the developer will always use it correctly
  - A contract you sign when you use C or C++

Fixes

- DO NOT USE MEMCPY() OR STRCPY()

- Alternatives (these do check bounds for you)
  - memcpy_s
  - strcpy_s

Even if you are a perfect coder...

If you can write perfect code…

- Let’s assume that you have gained the ability to write perfectly efficient and safe code
- Is your code always safe?

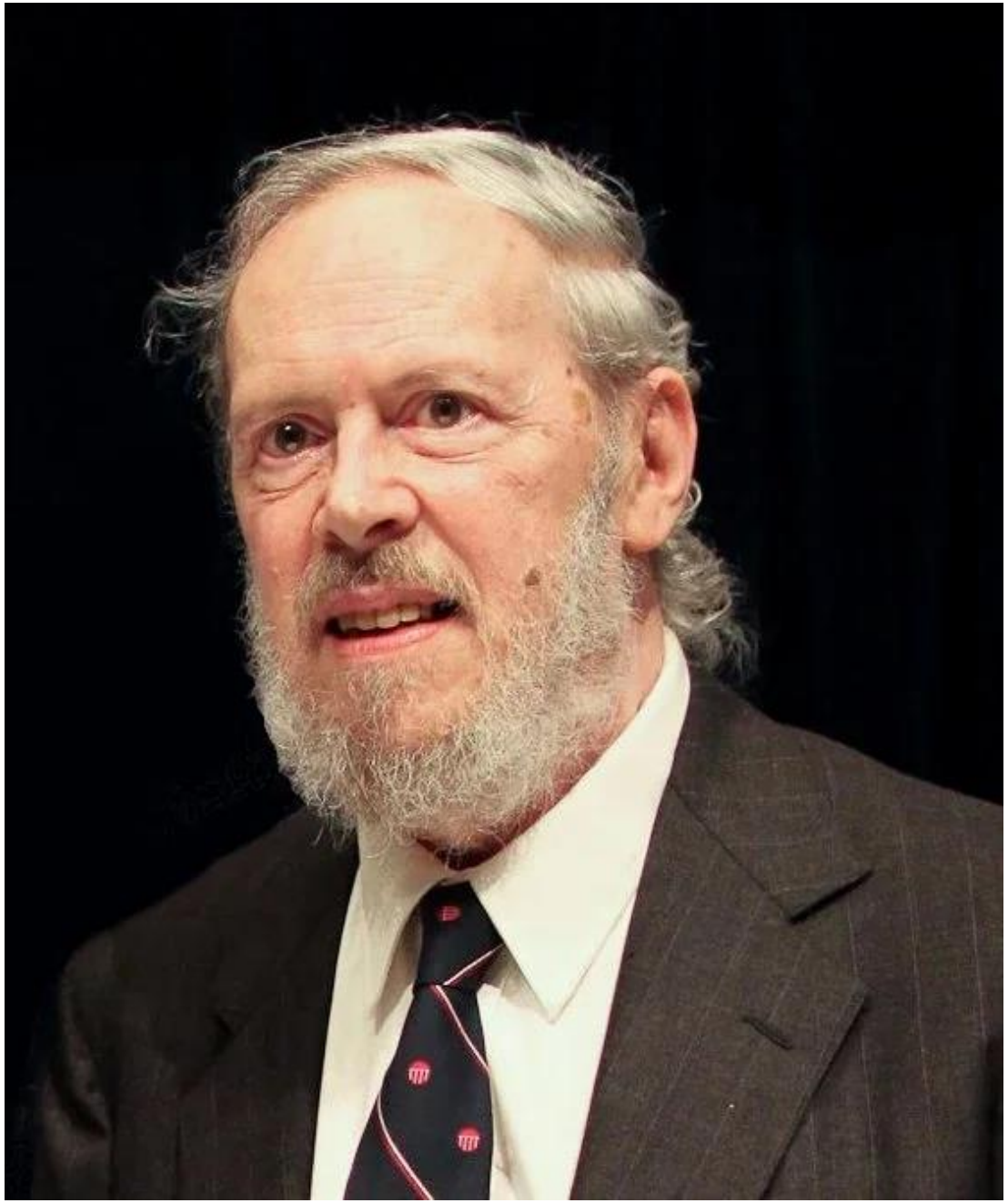
- What kind of packages do you import?

Log4J incident

- A widely used logging tool for Java
- A bug allowed for attackers to do Remote Code Execution
- A zero-day attack was first reported in November 2021 (CVE-2021-44228)[3]
- This vulnerability has been in the tool since 2013
- An estimated 93% of enterprise cloud environment was affected [4]
- Some services include: AWS, Cloudflare, iCloud, Minecraft servers, Steam, Tencent, etc.

Summary

- Low-level languages allow developers to directly manipulate memory
  - This allows for flexibility and efficiency, but also puts the burden on the developer to write safe and good code
- Vulnerabilities are not limited to code that you write, but code that you use and import as well
- There is no silver bullet to solve security, but require diligence from us (the devs) to think about types of attacks that can affect our code



Persons of the day

Dennis Ritchie,
Ken Thompson,
and Brian Kernighan
(and the rest of bell labs)

- Creator of the C language and a developer of the UNIX operating system
- Development started in 1970s in Bell labs
- How did C++ get its name?
- 1967 - Basic Combined

Programming Language (BCPL)

➔1969 - B [Ritchie and Thompson]

➔1972 - C [Ritchie and Kernighan]

➔1985 - C++ [Stroustrup]

References

- [1] https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html
- [2] https://nvd.nist.gov/vuln/detail/CVE-2014-1776
- [3] https://nvd.nist.gov/vuln/detail/cve-2021-44228
- [4] https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell