

Security and the Standard C++ Library

Martyn Lovell
Visual C++ Libraries
Microsoft Corporation



Agenda

- **Overview**
- The Problem
- Standard C++ Library Improvements
- Compile Time Diagnostics
- Runtime Diagnostics
- C Library Improvements
- Summary
- Q&A

Overview

- Software security is an industry-wide problem
- Microsoft has spent the last five years changing its culture
 - Changed our development processes
 - Secured our code and our platforms
 - Created new tools to detect problems
- Visual C++ 2005 [a.k.a. Whidbey] introduces a new generation of safer C++ development
 - Smarter compilers find problems before they happen
 - Safer libraries warn on unsafe constructs
 - Code checks at runtime for risky situations
- This talk includes
 - The philosophy and architecture of the changes
 - Examples of how to write safer code
 - Tips and tricks for working with the new libraries
 - Focus is the Standard C++ Library
 - Brief overview of C library changes too

Agenda

- Overview
- **The Problem**
- Standard C++ Library Improvements
- Compile Time Diagnostics
- Runtime Diagnostics
- C Library Improvements
- Summary
- Q&A

C/C++ History

- **Standard C initially evolved with Unix™**
 - Starting in 1970s
 - Public networking uncommon
 - Machines simpler, more constrained
 - First C language/library standard codified existing practice
- **Standard C++ evolved from C, adding power, abstraction**
 - Starting in the mid eighties
- **Standard C++ Library was a new departure**
 - Much safer and more powerful
 - Great abstraction and development model
 - Addition of STL to standard gave language a great foundation

A changed environment

- The arrival in the mid-nineties of widespread public networking changed everything
 - Now most code is under constant risk of attack
 - Users constantly cite security fears as among their most serious computing problems
 - Security issues have caused major problems
- **Developers must respond**
 - Code that was previously unimportant is now vulnerable
 - Assume every software component might be attacked
- The assumptions we used building code 20 years ago aren't appropriate any more
 - Users and networks can't be trusted
 - Networks can't be trusted
 - Must defend in depth
 - Must secure by default

Responding to the threat

- There is no silver bullet
 - Seriously
- To secure your code, the first thing that needs to change is you
 - Educate yourself
 - Realise your impact in creating security for your users
- Writing secure code impacts the whole process
 - Specification practice
 - Architecture and design
 - Implementation and debugging
 - Deciding when to ship
- This talk focuses on one part of the above
 - Don't assume that this is enough
 - These tools can help, but you still need the other steps

Libraries vulnerability

- **The standard C library is extremely vulnerable**
 - Encourages bad practice (strcpy)
 - Contains functions that can't be used safely (gets)
 - Lacks abstraction (direct manipulation)
- **By contrast, the Standard C++ Library was designed well from the start**
 - Encourages good practice (std::string)
 - Almost all functions have interfaces that can be safely implemented (std::copy, std::vector designed well)
 - Good abstraction provides appropriate places for additional checking [iterators, algorithms, containers]
- **But what we've learnt tells us that even C++ code needs to change**

Agenda

- Overview
- The Problem
- **Standard C++ Library Improvements**
- Compile Time Diagnostics
- Runtime Diagnostics
- C Library Improvements
- Summary
- Q&A

What we've learnt

- Over the last several years, Microsoft has learnt a lot about security
 - Understand what we did wrong in the past
 - Reengineered our processes
 - Rearchitected our products
 - Detected several common aniti-patterns
 - Reviewed and secured our code
- This generation of our products
 - Help developers benefit from what we learnt
 - Provides tools we use internally to find problems
 - Helps you make your code safer with minimal change

Some key security lessons

- No silver bullet
 - Need to address people, process, code, product
- Humility
 - no matter how often you have shipped code, or how smart the developers, you have subtle bugs
- Safe defaults
 - Products need to install in their safest mode
- Defend in depth
 - Don't just attempt to secure outer boundary
- Mitigate risk
 - Don't just fix bugs
 - Remove sources of potential bugs

Safety in the C++ Libraries

- **Key goals**
 - Warn on risky constructs
 - Find problems at compile time
 - Minimize code change
 - Minimize performance impact
- **Mitigate most relevant risks**
 - Buffer overruns
 - Invalid data
- **C++ Library makes this easier**
 - More abstracted
 - More modern
 - Requires less change
 - Optimised for minimal abstraction penalty

Safety improvements

- Compiler warnings on lines where risky constructs are used
- New compiler option for (slow) deeper analysis to find potential bugs
- New debug-build runtime checks
- New release-build safety checks to ensure no unexpected outcomes
- More detail on each of these follows

Agenda

- Overview
- The Problem
- Standard C++ Library Improvements
- **Compile Time Diagnostics**
- Runtime Diagnostics
- C Library Improvements
- Summary
- Q&A

Unsafe code warnings

- By default, we warn on risky lines of code
- Generally results in a small number of warnings for large codebases
- We use these warnings internally to secure our code
 - Warnings indicate use of constructs we consider too risky for shipping code
 - Warnings describe that a construct is deprecated, and reference documentation
 - This is not “deprecated” in the C++ standard sense
- Minimising risk means defensive coding
 - If code has enough information to check itself at compile or runtime, it is much safer
 - Not all risky code has bugs
 - But risky code encourages later bugs because it lacks the ability to self-check its input.

Risks vs bugs

- This code has a bug

```
void helloworld(void)
{
    std::wstring text(L"Hello world");
    wchar_t buffer[5]=L"";           // bug
    std::copy(
        text.begin(),
        text.end(),
        buffer
    );                               // overrun
    std::wcout << buffer;
}
```

Risks vs bugs (2)

- This code does not have a bug, yet...

```
// Original
void helloworld(void)
{
    std::wstring text(L"Hello world");
    wchar_t buffer[15]=L"";           // big enough, right now
    std::copy(
        text.begin(),
        text.end(),
        buffer
    );                               // safe, but risky
    std::wcout << buffer;
}
```

- Our compiler would have warned even so
- Because `std::copy` can't know it is safe
 - It doesn't know how big buffer is

Risks vs bugs (3)

- This code is safe, but not bug-free

```
void helloworld(void)
{
    std::wstring text(L"Hello world");
    std::wstring buffer; // bug - not enough space
    std::copy(
        text.begin(),
        text.end(),
        buffer.begin()
    ); // safe - iterator is checked
    std::wcout << buffer;
}
```

- Use of a `std::string` for output ensures that the iterator passed to `copy` knows how much space is available
- Iterators are checked by default
- We can warn at run-time about the bug (lack of space)

Risks vs bugs summary

- **Warnings only fired if code is unsafe**
 - `std::copy` works fine with iterators
 - Use with pointers elicits warning
- **It's not enough to fix bugs**
 - Everyone has bad days, and all teams have time and resource pressure
 - Unrelated external changes can expose old code to new risks
 - We recommend general policy to reduce introduction of risky code
- **Every time we warn, it's a problem we would have fixed**
 - We have dogfooded these warnings extensively
 - We know that risky constructs create problems

When do we warn

- Use of output iterators with unknown size
 - Generally, pointers
- Use of a small number of specific C++ functions
 - which take output data without providing a size
- On each line where a risky construct occurs

How to respond

- **To disable C++ standard library warnings**
 - Because you don't want to respond right now
 - use `/D_SCL_SECURE_NO_DEPRECATED`
 - Similar controls exist for C, ATL, MFC
- **Medium term**
 - Change code to be non-risky
 - For example, use a `std::wstring` instead of a `wchar_t` buffer
 - **Portable solution**
 - or use a `stdext::checked_array_iterator`
 - More on checked iterators later

Static Code Analysis

- New feature for Visual Studio Team System 2005, Developer Edition
- Lint-on-steroids
- Focussed especially on finding security issues
- Runs static-analysis of whole function to determine potential risk paths
- CI /analyze (also on project menu)

Static code analysis

- Consider this classic bug

```
void allocate(void)
{
    wchar_t *str=new wchar_t[8];
    memset(str, 0, sizeof(str)*sizeof(wchar_t));

    // stuff
}
```

- It generates a warning with `cl /analyze`

Code annotation

- You can teach code analysis more about your functions
- This allows the analyzer to find more problems.
- For example this declaration
 - `__bcount(_Size) void *operator new(size_t _Size);`
- ... explains that operator new returns `_Size` bytes
 - Full list of annotations in `sal.h`
- All our standard functions are annotated
- This found many real bugs in our code

Agenda

- Overview
- The Problem
- Standard C++ Library Improvements
- Compile Time Diagnostics
- **Runtime Diagnostics**
- C Library Improvements
- Summary
- Q&A

Goals

- Improve safety in shipping builds
- Automatic checking for standard libraries
- Minimal performance impact in most scenarios
- Fit in with Standard C++ Library architecture
- Provide extra deep checking in debug

Checked iterators

- Present in both retail in debug builds
- Checked iterators have enough context to validate
 - They know their host container
 - Can use to ensure that write operations do not go past end of container
- Checks occur on increment, dereference, creation
- These iterators have been very successful at finding bugs
- Standard-conformant, and enabled in standard types (`std::vector`, `std::string`, ...)
 - `std::vector::begin` still returns a `std::vector::iterator`
 - `std::vector::iterator` is checked

Checked iterators for memory

- Can provide checked iterators for raw memory containers
 - `stdext::checked_array_iterators`
 - Extension to standard
- Because of optimisations described later, mostly perform as well as raw memory
- Ensure that even raw memory benefits from compile and runtime checking
- Code explicitly states size of memory (in iterator constructor) allowing constraint checks

Memory checked iterators

```
// Original
void helloworld(void)
{
    wchar_t text[]=L"Hello";
    wchar_t buffer[20]=L"";
    std::copy(
        text,
        text+_countof(text),
        buffer
    ); // compiler warning
    std::wcout << buffer;
}
```

```
// checked iterator
using namespace stdext;
using namespace std;
void helloworld2(void)
{
    wchar_t text[]=L"Hello world";
    wchar_t buffer[20]=L"";
    checked_array_iterator <wchar_t *>
        dest(
            buffer,
            _countof(buffer)
        );

    copy(
        text,
        text+_countof(text),
        dest);
    wcout << buffer;
}
```

Our previous example

- This code is safe, but not bug-free

```
void helloworld(void)
{
    std::wstring text(L"Hello world");
    std::wstring buffer;                               // bug - not enough space
    std::copy(
        text.begin(),
        text.end(),
        buffer.begin()
    );                                               // safe - iterator is checked
    std::wcout << buffer;
}
```

knows how much space is available

- Iterators are checked by default
- We can warn at run-time about the bug (lack of space)

C++ Runtime check

```
// Why checks are in iterators
void pointless(void)
{
    std::wstring str(L"Hello world");
    std::wstring::iterator iter=str.begin();
    for(int i=0; i<20; i++)
    {
        *iter=L'a'; // bug, runtime check invoked
        iter++;
    }
}
```

Performance impact

- Checks at each iterator usage add cost
 - Optimiser can sometimes remove checks
- Key cost-point is inside algorithms
 - Where data structures get high-touch access
- Most checks happening during algorithms are redundant
 - `std::copy` doesn't need to check at each step, just at start
- So checked algorithms do checking upfront and then use unchecked iterators
 - Makes them almost as fast as their unchecked cousins
 - Side effect
 - `std::for_each` faster than similar `for()` loop

Performance impact

- **Iterator size**
 - Iterators get larger to remember container
 - Only impacts code with very large number of long-lived iterators
 - Rarely a visible issue
 - **Consider using unchecked iterators for broad storage**

Debugging improvements

- Great Dinkumware innovation we've picked up in Visual C++ 2005
- Much deeper checking in debug builds
 - Checks iterator relationships, container relationships, more conformance rules
 - Issues debug errors at runtime
- Has been great at finding bugs
- Performance impact higher, but generally consistent with other debug features

Migration Path

- Can be disabled with `/D_SECURE_SCL=0`
- Generally can be enabled with very low impact for code
 - Just finds a few bugs when enabled in existing code
- Few code changes needed
 - Perhaps adjust algorithms to check up-front
 - Switch to use of `for_each` in central cases
- Debug checks can be disabled too if needed
 - `/D_HAS_ITERATOR_DEBUGGING=0`

When a check fails

- Your program has a severe bug
- It was about to buffer-overflow
- This indicates a severe internal logic error
- There is no safe way to recover
 - String truncation can be unsafe
- Debug – invoke debugger
- Retail - Two choices
 - Abort program [default] – report fault to OS “Watson”
 - Throw `std::exception`
- Abort process default is consistent with other safety features
 - /GS, operating system
- Continuing to run destructors with an unknown state can be risky
 - For example, imagine that a different security context was active, and destructors then run in the wrong context

Standardisation

- **The C parts of this work are in process of becoming a technical report to the C standard**
 - Working with the committee since early 2003
 - Extremely useful and productive process
- **Too early to standardise the C++ parts**
 - Need to get experience of broad deployment first
 - Want to work with the C++ committee in future on these issues
 - Presented outline of plans to committee in 2003
- **C++ innovation is standard-conformant**
 - New classes not in standard namespace
 - Flagged behaviours already undefined
 - Unsafe code warnings do not stop object generation
- **Some standard extensions are required**
 - `stdext::checked_array_iterator`, for example
- **This first release should allow us to gain implementation experience**
 - Later standardisation would help portability

Agenda

- Overview
- The Problem
- Standard C++ Library Improvements
- Compile Time Diagnostics
- Runtime Diagnostics
- **C Library Improvements**
- Summary
- Q&A

C Library Changes

- C Library problem is more severe
 - Many unsafe practices
- Safest solution
 - Move to a well-abstracted class
 - `std::string` instead of `strcpy`
- Not practical for many users
 - Too much impact
 - Too much code churn
- So our goals are to
 - improve existing code without radical change
 - warn on risky practices
- Based on internal work we've used for several years

Key security problems

- Buffer overruns
- Error reporting & validation
- Parameter validation
- File access rights
- Static buffer results
- File paths & permissions

Interface problems

- **Lack of buffer size**
 - strcpy
- **Callback context needed**
 - Avoid static variables
 - Safe for reentrancy, threads
 - qsort, bsearch
- **Never use static result buffers**
 - Can overrun
 - tmpnam

Definition problems

- **Returning unterminated strings**
 - Predictable, but hard to get right
 - strncpy, sprintf

Implementation problems

- **Parameter validation**
 - Ensure errors reported
 - Provide way to catch errors
 - We will have handler function
 - assert in “debug” build
- **Stack usage**
 - Avoid excessive usage
 - Stack overflow can be a denial of service attack

Implementation problems

- **File permissions**
 - Create temporary files safely
 - Create all files by default with good permissions
- **Long file path support**
 - Don't fail when given extended paths
 - Windows specific

Diagnostics

- Diagnose use of risky functions
 - strcpy, gets, etc
- Diagnostic describes function to use instead
- C++ templates used to automatically 'fix' problems where possible
 - strcpy to static buffer converted to strcpy_s
- Annotated declarations allow /analyze to find many problems
- Execution time checks and validations

Summary

- We have learnt that code needs the ability to check itself at compile and run time
 - Our library allows this to happen, often with minimal code change
- Security is a critical priority
 - these libraries can help improve it
- But they are not a silver bullet

Questions?

- Great reference
 - Writing Secure Code (Howard/Le Blanc)
- Ask me anything about Visual Studio 2005
- Follow-up:
 - <http://blogs.msdn.com/martynl>
 - MartynL@microsoft.com

Backup

Example function

- **Old:**

- `size_t mbstowcs`
(
 `wchar_t *wcstr,`
 `const char *mbstr,`
 `size_t count`
);

- **New:**

- `errno_t mbstowcs_s`
(
 `size_t *pConvertedMBChars,`
 `wchar_t *wcstr,`
 `size_t sizeInWords,`
 `const char *mbstr,`
 `size_t count`
);

Example code change

```
// Original
wchar_t dest[20];
wcscpy(dest, src);
wcscat(dest, L" ...");
wprintf(L"%s", dest);
```

```
// Basic remediation
wchar_t dest[20];
wcscpy_s(dest,
          _countof(dest), src);
wcscat_s(dest,
          _countof(dest), L"
          ...");
wprintf(L"%s", dest);

// Abort if we don't fit
// assumes 20 really is a
// good size
```

Version 2

```
// Original
wchar_t dest[20];
wcscpy(dest, src);
wcscat(dest, L" ...");
wprintf(L"%s", dest);
```

```
// better
wchar_t dest[MAX_SIZE];
wcscpy_s(dest,
    _countof(dest), src);
wcscat_s(dest,
    _countof(dest), L"
    ...");
wprintf(L"%s", dest);
// Now we know we can fit
// if MAX_SIZE is right
```

Version 2a

```
// Original
wchar_t dest[20];
wcscpy(dest, src);
wcscat(dest, L" ...");
wprintf(L"%s", dest);
```

```
// better
// rely on templates

wchar_t dest[MAX_SIZE];
wcscpy_s(dest, src);
wcscat_s(dest, , L" ...");
wprintf(L"%s", dest);
```

Version 2b

```
// Original
wchar_t dest[20];
wcscpy(dest, src);
wcscat(dest, L"
    ...");
wprintf(L"%s", dest);
```

```
// better
// rely on templates

wchar_t dest[MAX_SIZE];
wcscpy_s(dest, src);
wcscat_s(dest, L" ...");
wprintf(L"%s", dest);

// size parameters still needed
// for non-arrays
```

Version 2c

```
// Original
wchar_t dest[20];
wcscpy(dest, src);
wcscat(dest, L" ...");
wprintf(L"%s", dest);
```

```
// better
// standard templates
// off by default
#define
    _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES

// off by default
wchar_t dest[MAX_SIZE];
wcscpy(dest, src);
wcscat(dest, L" ...");
wprintf(L"%s", dest);

// Still need wcscpy_s for char
*
```

Version 3

```
// Original
wchar_t dest[20];
wcscpy(dest, src);
wcscat(dest, L" ...");
wprintf(L"%s", dest);
```

```
// Perhaps we
// have time for bigger
// change

std::wstring dest(src);
dest+=L" ...";
cout << dest;
```