Hardening Your Code

Marshall Clow
Qualcomm, Inc.
marshall@idio.com
Twitter: @mclow
Blog: http://cplusplusmusings.wordpress.com
(occasional)

ACCU April 2015

It's a dangerous world

- Active attackers
- Environments that you did not envision
- Good methodology is not enough
  - Mistakes can happen

What can you do to increase your confidence in your code?

- Remember your history
- Test, test, test
- Look at compiler warnings
- Static analysis
- Dynamic analysis
- Fuzzing

Version Control

- Remember how you got where you are
- You can go back in time
- Remembers releases

A War Story

- http://esr.ibiblio.org/?p=6205
- Summary
  - Code stopped working (tests failed)
  - Reverted last change; still not working
  - Used ‘git bisect’ to find breaking change

Automated Tests

- Have a test suite
- Run it often
- Add to it whenever you can

But .. I don't have automated tests
(and that sounds like a lot of work)

- Write some tests (even if it's just one)
  - Run it/them! (often)
- A small test suite is better than no test suite
- Add to it whenever you can.
- Don't let failing tests fester.

What kind of things should I test?

- Normal operations
- Edge cases
- Error conditions

When you write unit tests, TDD-style or after your development, you scrutinize, you think, and often you prevent problems without even encountering a test failure.

-- Michael Feathers *The Flawed Theory Behind Unit Testing*

http://michaelfeathers.typepad.com/michael_feathers_blog/2008/06/the-flawed-theo.html

Someone reports a bug. What do you do?

- Write a test that reproduces the bug.
- Add this to your automated tests
- Run the test; verify that it fails
- Implement a fix
- Verify that the test passes (and all the other tests, too!)

Why are tests important?

- They give you confidence in your code
- They give you the ability to change your code w/o fear.
  - “What if I break something?”
  - Then the tests should catch it
  - Enables refactoring

Compiler Warnings

Who cares about compiler warnings?

- You should.
- The compiler is telling you “there’s something here in your code that isn’t what I expect”
- If you have lots of warnings, it’s really hard to tell when you get a new one.

Toy Example #1

```
unsigned test (unsigned foo) {
  if (foo >= 0)
    return foo;
  return 123;
  }
```

Test with different compilers

- Different compilers have different sets of warnings
- Staying warning-free on multiple compilers can be hard.

When I started working on *[Product]*, there were tens of thousands of warnings in the source code in whatever version of gcc was in Xcode 2.1.

We are now (and have been for almost five years) warning free. With only a dozen or so uses of manually shutting off a warning around a block of code using the pragma for clang, and with maybe a hundred similar pragmas on Windows (because the Windows system header files produce warnings), and we're now running with the default warnings that are enabled in Xcode 5.1, which is significantly stricter than gcc ever was.

How did we do that? By turning on warnings (and treat warnings as errors) and going through and clearing up every single one of them. Took a couple years of me and one other engineer working on it in our "spare time" to get to zero, but staying at zero is significantly easier than getting there was.

Static Analysis

- Multiple commercial products
- Some open-source checkers as well

Toy Example #2

```
char *get_string (int foo) {
  if (foo == 0)
    return NULL;
  return "123";
  }

int len (const char *s) {
  return strlen(s);
  }

int bar = len(get_string(x));
```

More about static analyzers

- They find “interesting” bugs.
- They are heavyweight tools
  - Expensive to run
  - Expensive to purchase
- They tend to have high false positive rates

Dynamic Analysis

- Build an instrumented version of your program
  - You don't ship this version
- Test it
- Report when things go wrong

Examples

- Assertions
- “Debug mode”
- “Sanitizers”

Sanitizers

- Usually implemented by compiler vendor
  - Address
  - Undefined Behavior
  - Memory
  - Thread

Sanitizers (2)

- Very few (goal: 0) false positives
- Report misbehavior when it happens
  - Can report stack crawl, etc.

Using ASAN

- http://blog.llvm.org/2013/03/testing-libc-with-address-sanitizer.html
- Ran ~4350 tests; found three bugs
  - One bug in the library being tested
  - Two bugs in the test code
- https://www.mozilla.org/security/announce/2014/mfsa2014-49.html

Undefined Behavior

- What is Undefined Behavior?
  - Integer overflow
  - indirecting through NULL
  - indexing off the end of an array
  - Other things.

UBSAN

- Catches undefined behavior
- Examples
  - “load of value 123, which is not a valid value for type ‘bool’”
  - “runtime error: index 40 out of bounds for type 'char_type [10]'”
- http://blog.llvm.org/2013/04/testing-libc-with-fsanitizeundefined.html

Code Coverage

- Records which parts of your code get executed.
- Another reason to have a good test suite

Example #1

```
2034             : template <class _Tp, class _Allocator>
2035             : bool
2036             : vector<_Tp, _Allocator>::__invariants() const
2037             : {
2038          51 :     if (this->__begin_ == nullptr)
2039             :     {
2040          24 :         if (this->__end_ != nullptr || this->__end_cap() != nullptr)
2041           0 :             return false;
2042          12 :     }
2043             :     else
2044             :     {
2045          39 :         if (this->__begin_ > this->__end_)
2046           0 :             return false;
2047          39 :         if (this->__begin_ == this->__end_cap())
2048           0 :             return false;
2049          39 :         if (this->__end_ > this->__end_cap())
2050           0 :             return false;
2051             :     }
2052          51 :     return true;
2053          51 : }
2054             :
```

Example #2

```
1421             :
1422             : template <class _Tp, class _Allocator>
1423             : void
1424             : vector<_Tp, _Allocator>::assign(size_type __n, const_reference __u)
1425             : {
1426        2855 :     if (__n <= capacity())
1427             :     {
1428        1718 :         size_type __s = size();
1429        1718 :         _VSTD::fill_n(this->__begin_, _VSTD::min(__n, __s), __u);
1430        1718 :         if (__n > __s)
1431           0 :             __construct_at_end(__n - __s, __u);
1432             :         else
1433        1718 :             this->__destruct_at_end(this->__begin_ + __n);
1434        1718 :     }
1435             :     else
1436             :     {
1437        1137 :         deallocate();
1438        1137 :         allocate(__recommend(static_cast<size_type>(__n)));
1439        1137 :         __construct_at_end(__n, __u);
1440             :     }
1441        2855 : }
1442             :
```

Fuzzing

- Take a set of valid inputs, and "modify" them.
- Feed the modified inputs into the program, and look for misbehavior
- http://blog.chromium.org/2012/04/fuzzing-for-security.html

American Fuzzy Lop

- New open-source fuzzer
  - http://lcamtuf.coredump.cx/afl/
  - Released last October
- Uses code coverage and behavior analysis to guide itself.
- "Pulling JPEGs out of Thin Air"
  - http://lcamtuf.blogspot.co.uk/2014/11/pulling-jpegs-out-of-thin-air.html

Fuzzing and Dynamic Analysis

- These two techniques work *very* well together.

Wrapping up

- All of these things can be started small
  - Except maybe the static analysis
- Improve them over time
- They work well together (tests and source control, Sanitizers and fuzzing, etc)

Questions?