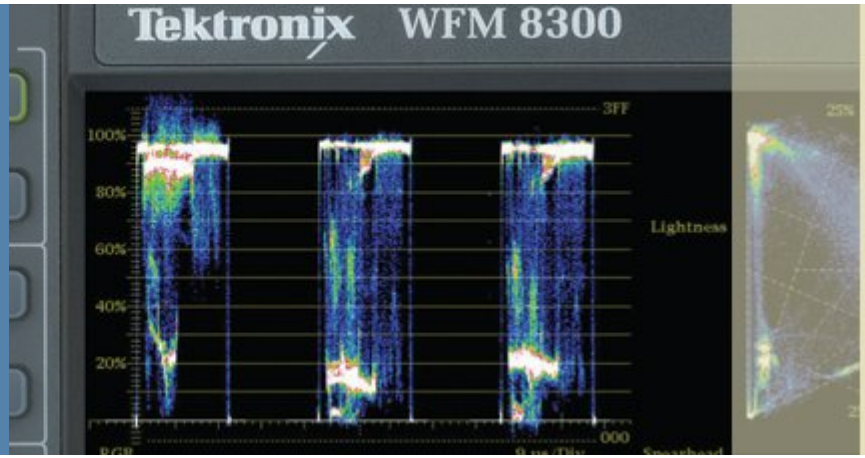
Using Static Code Analysis to Find Bugs Before They Become Failures



Presented by Brian Walker

Senior Software Engineer,
Video Product Line, Tektronix, Inc.

Pacific Northwest Software Quality Conference, 2010

In the Beginning, There Was Lint

- Syntactic analysis found many simple coding mistakes
  - Mismatched arguments
  - Incompatible data type usage
  - Uninitialized variables
  - Printf() format argument mismatch
- Analyzed one source file at a time
- Generated lot's of warnings
  - Not all of them useful
- Function is now integrated into most compiler

Beyond Lint

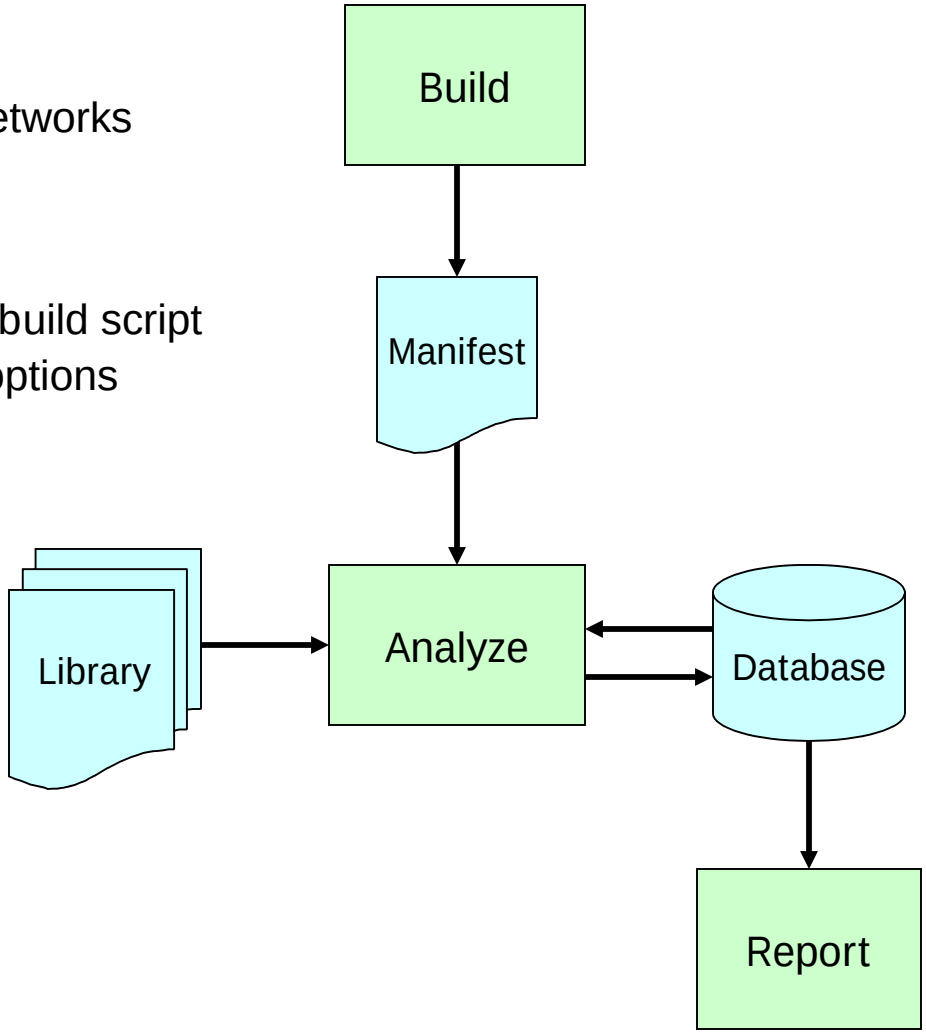
- Functional analysis without execution
  - Functional analysis, not just compilation errors
  - Data flow and control flow analysis
  - Typically automated
- Incorporates some aspects of reverse engineering
  - Analyzes objects and their usage
  - Follows resource allocation, initialization and de-allocation
- Considers the whole program
  - Analyses entire build, not just individual files
  - Examines all execution paths within functions and between functions
  - Calculates range of possible values for each variable
  - line by line, complete coverage (for known issues)
- Quality, not quantity
  - Balance aggressiveness with restraint

Elements of a Static Code Analysis System

- Library of issues and patterns
  - Identifies problem severity
  - Ability to define new patterns or suppress unwanted patterns
- Database for tracking issues
  - Manage issue priority and ignore false positives
  - Identify new issues over time and track as source files change
  - Identify fixed issues
- Accessible issue reporting interface
  - Detailed description of specific issue and location in source code
  - Assist developers to understand, investigate and fix bugs
  - Sometimes integrated into IDE
- Compilation process to perform analysis
  - May require as much time as actual build
  - Must be run with the same options and compiler flags

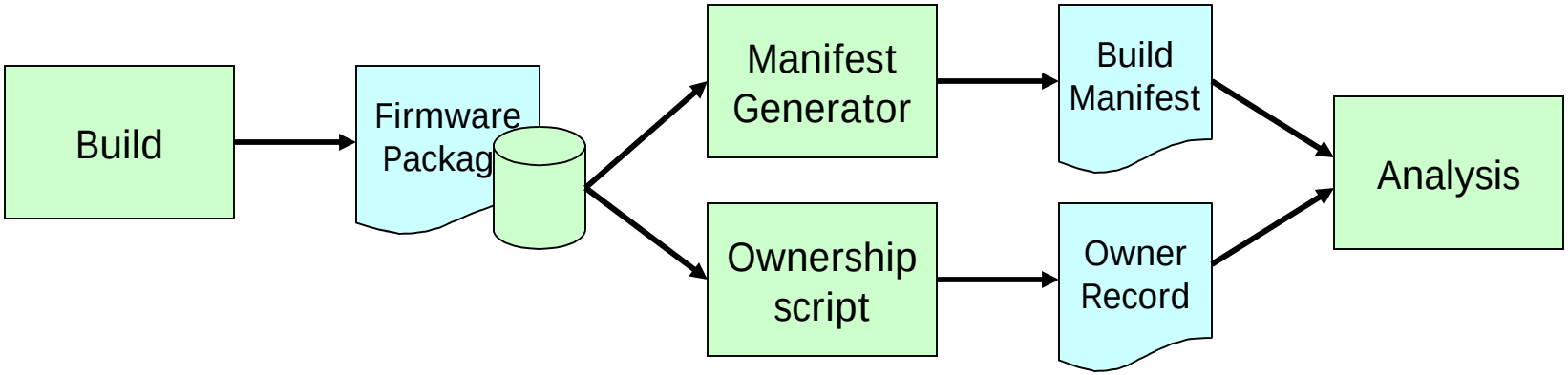
Using the Klocwork Static Analysis Tool

- Klocwork Insight
  - Software developed at Nortel Networks
  - Analyzes C, C++, C# and Java
- Uses build auditing approach
  - Manifest generator runs normal build script
  - Captures files, commands and options
  - Generates manifest
  - Directs analysis
  - Consults library
  - Queries database
  - Updates database
  - Generates reports



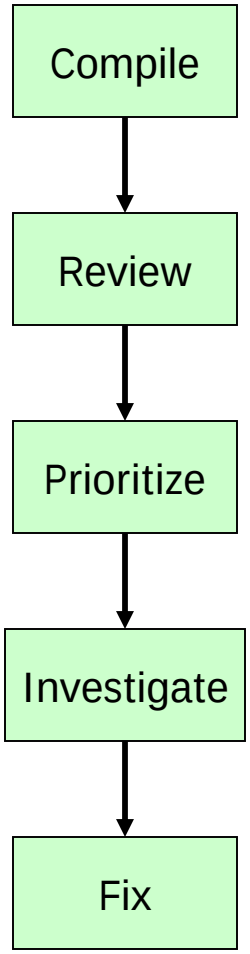
Configuring the Analysis

- Klocwork does not support incremental builds
  - Needs to know all of the files in the build, needs to be reminded
  - Vendor suggested rebuild the manifest after adding/removing files
  - or, Perform full build every time
- Integration with ClearCase
  - ClearCase records complete configuration record for build
  - Build manifest file format clearly described in Klocwork documentation
  - Developed simple script to create manifest from ClearCase config record
  - Generate file ownership from version history



Integrating Static Analysis in our Environment

- Incorporated into nightly builds
  - Klocwork analysis runs as a step in the nightly build
  - Uses ClearCase configuration record to produce manifest
  - Uses ClearCase version history to determine file ownership
- Manual review of new issues reported
  - Review of new and uncited issues
  - Set status according to severity and priority
  - Forward new issues to team members
- Investigate and make changes to source code
  - Analysis usually provides enough detail to quickly fix issue
  - Use browser to search for declarations and implementation



Managing Issues in Klocwork

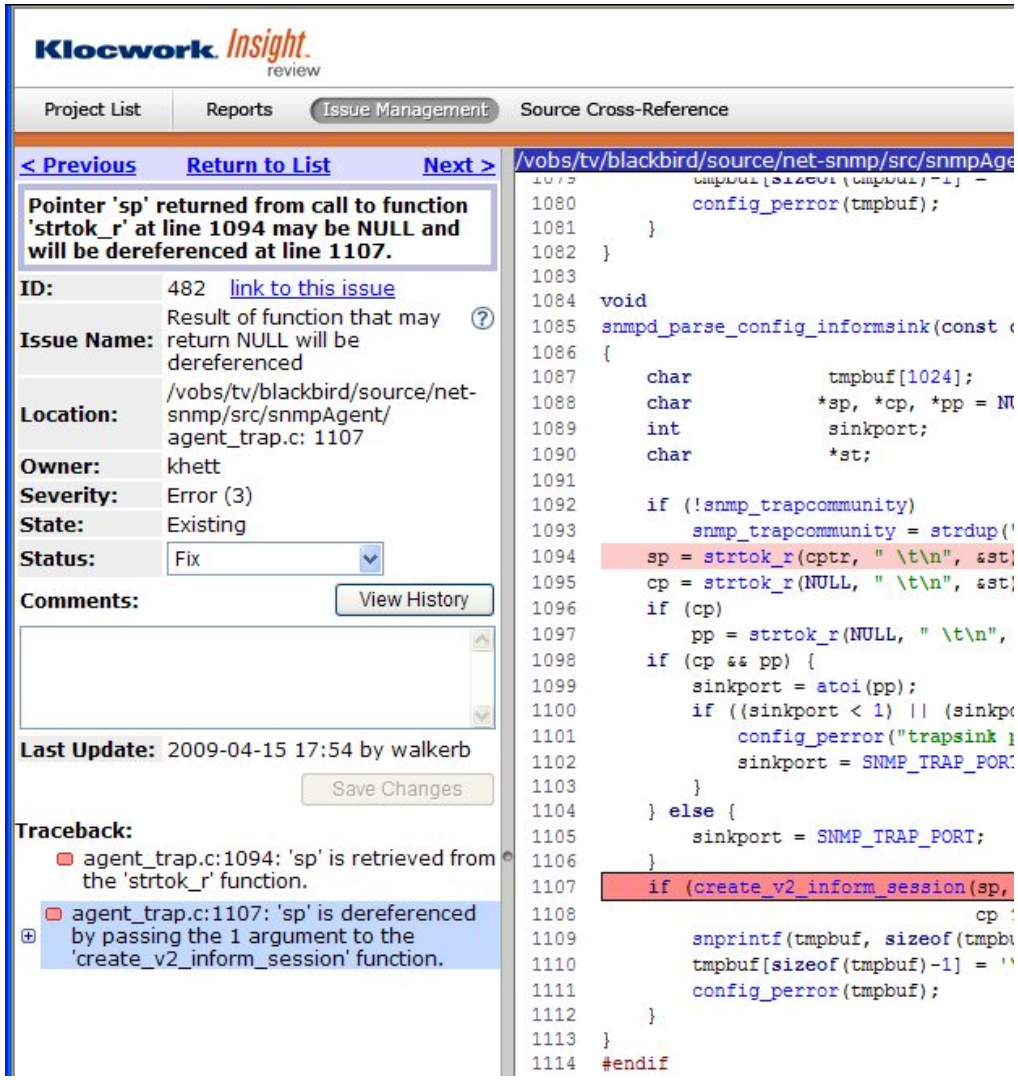
- Klocwork determines the “state” of an issue
  - Identifies issues as new, existing, recurred, fixed or not in scope
  - Maintains history of current issues
  - Detects when new issues are discovered
- Reviewer determines the “status” of an issue
  - Analysis -> Citing -> Resolution
  - Every new issue starts with status set to analyze (uncited)
  - Users choose fix, fix in next release, fix in later release, not a problem
  - Reviewer and implementer can add notes to issue record
  - Analyze or Fix status identifies open issues
- Issues with issue management process
  - Would be nice if citing could automatically generate bug report
  - Does not provide real tracking capability

Klocwork Issue Reports

- Contains a specific description of the problem
  - “Array ‘foo’ of size 22 may use index values or -2 .. -1. Also there are similar errors on lines 491, 494.”
  - Provides traceback of analysis showing all locations that affect value
  - Traceback references highlighted in source code listing
  - Nested references can be expanded for deeper dive
- Cross reference interface allows exploration of source code
  - Finds declarations functions, variables, macros and constants
  - Finds where items are used, referenced, read or written
  - Welcome feature to aid in the investigation of issues

Klocwork Issue Detail Display

- Issue description
  - Summary
  - Link to record
  - Name
  - Location
  - Owner
  - Severity
  - State
  - Status
  - Comments
- Traceback of problem path
  - Listed in order of execution
  - References to lines of code
  - Nested references to other functions
  - Nested references expand for deeper investigation



Klocwork Issue Identification

- Klocwork uses short hand notation to identify problem types
  - ABR: Array Buffer Overrun
  - NNTS: Non-Null-Terminating String
  - Tends to be concise and memorable
  - Can often determine type of problem in a glance
- Online documentation tends to be well written and informative
  - Provides description, specific vulnerabilities, risks, mitigation and prevention
  - Focus on security
- One common distinction: must vs. might
  - “Must” issues indicate error that occur as a result of a single failure
    - NPD.FUNC.MUST: NULL pointer returned by function always dereferenced
  - “Might” indicates a failure might occur if more than one condition is true
    - NPD.FUNC.MIGHT: NULL pointer dereferenced under certain conditions

Types of Issues Found by Static Analysis

- Null pointer dereferences
  - Very effective in discovering null pointer dereference issues
  - Many due to not checking pointer parameters for NULL
  - NPD.CHECK.MIGHT: pointer checked for NULL but still dereferenced
  - Can be quite insidious and may be overlooked in code review

```
int handle_request( int op, int reqid, PduType *pdu, SyncState *state)
{
    if (reqid != state->reqid && pdu && pdu->command != MSG_REPORT) {
        return 0;
    }
    if (op == RECEIVED_MESSAGE) {
        if (pdu->command == MSG_REPORT) {
            blah;
            blah;
            blah;
        }
    }
```

Types of Issues Found by Static Analysis

- Array buffer overruns
  - Klocwork has discovered quite a few flavors
  - String handling is common theme, especially regarding numbers

```
char buffer[8];
sprintf( buffer, “%s%d”, type, value);
```

- Memory leaks
  - Often caused by improper error handling
  - Often not covered by functional testing
  - Might be covered by unit testing
  - C++ class constructors and destructors
    - Expects that memory allocated in a constructor is freed in destructor
    - Also expects copy constructor and assignment operator
  - Death by a thousand pin pricks

Types of Issues Found by Static Analysis

- Use of freed memory
  - Checks for use of memory pointer after return to system heap
  - Did not expect to see in our software
  - Found in third-party and open source components
- File Resource Leaks
  - Like memory leaks, often the result of improper error handling
  - Klocwork reports as lower priority
  - Have found instances where file handles can be lost
- Concurrency violations
  - Our software is often multithreaded
  - Uncovered problems with semaphore handling
    - Warns when semaphores taken but not returned within a function
    - Warns when thread goes to sleep while still holding semaphore

Types of Issues Found by Static Analysis

- Security vulnerabilities
  - Becoming a greater concern as more devices become connected
  - Secure programming practices less obvious and often overlooked
  - Vulnerable program will still pass functional tests
  - Testing requires effort to actively exploit vulnerabilities
- Number of reported vulnerabilities continues to grow
  - Risk of failure includes potential for exposure of sensitive data
  - Prevention is far more effective than remediation
- Most vulnerabilities can be traced to a relatively small number of programming errors
  - Klocwork references Common Weakness Enumeration dictionary
  - Klocwork recently added support for MISRA C and C++ coding standards
    - Adopted by aerospace, telecommunications, medical, defense and others

Analyzing the Analysis

- Finds many high-quality issues in the source code
  - From 500,000 lines of C/C++ in more than 2000 files
  - Found 1500 issues
  - Rejected 175 issues
- Somebody had to look at those 1500 Issues
  - Established code base, and third-party contributions
  - Examine, prioritize and eventually fix
- True value seen after bug debt has been paid
  - Problems found can be quickly identified and corrected
  - Most useful when employed on new software as preventative

Analyzing the Analysis

- Potential to save many hours of frustration
  - Hours spent trying to reproduce and capture and instance of a crash bug
  - Race conditions can be the most devious and time consuming
  - Found in Klockwork database after the fact.
- Other bugs are more treacherous
  - Array access may overflow into unrelated data structure
  - May cause mayhem in unrelated thread
  - Or write over a virtual table causing mysterious crash
  - Debugging requires setting watch point to catch in the act
- But, not perfect
  - Has reported at least one false positive
  - Has missed at least one array buffer overrun that caused random crash
  - Ideally maintains a balance between useful and annoying
- Promotes good programming practices

Looking Forward

- Desktop integration
  - Potentially provide instant feedback to developers
  - Developers could fix errors before committing to version control system
  - Many vendors provide integrations for Eclipse and Visual Studio
  - Still perform bulk of analysis nightly
- Compiler integration
  - Some vendors provide static analysis in compiler
  - Integrated with build processes
  - Limits choice of compiler

Fitting it all Together

- Excellent tool for finding bugs early in development
  - Compliments compiler warnings
  - Does not require actual hardware to analyze software
- Complements dynamic analysis (functional testing)
  - Able to follow all/most paths within each function
  - Testing and analysis are adept at finding different kinds of faults
    - Cannot test functional requirements through static analysis
    - Cannot reproduce all scenarios through emulation
- Complements formal code review and peer review
  - Consistently finds instances of known issues
  - Cannot replace expert eyes of peers
- Best option for testing third-party and open source software
  - Most open source software is provided “as-is and without warrantee”
- Increases the tribal knowledge of an organization

Questions and Answers

Brian Walker

Brian.R.Walker@tektronix.com