Static Analysis on Steroids: Parasoft Data Flow Analysis for C and C++

Static Code Flow Analysis – Introduction

The term *static code analysis* means different things to different people in the software industry. In this paper, static code analysis is defined as analysis of source code without execution. There are two principle approaches to static analysis: (1) pattern-based analysis and (2) program execution or flow-based analysis.

Pattern-based analysis supports implementation of a set of coding practices, or a coding policy. For C++, many such "best practices" are defined in reference books [1] and de facto industry standards such as Ellemtel, MISRA, and common guidelines for 32- and 64-bit portability. In addition to supporting the common C++ guidelines, static analysis tools should also allow you to customize rules to suit your specific application and implementation context, as well as help you identify code patterns indicative of application-specific defects.

Flow-based analysis is the technique of logically executing the program to track the propagation of data values, their effects on control flow, and the legality and cleanliness of data at multiple points in the code. Flow analysis attempts to simulate the runtime condition of data objects across functions, modules, and files. The goal is to uncover code problems such as memory corruption, leaks, invalid pointer dereferences, unsafe or tainted data propagation, and security vulnerabilities. Whereas pattern-based analysis focuses on looking for local syntactical anomalies, flow analysis explores the potential execution paths of the larger code context. This bug-prevention technique is powerful because it does not depend on user input to identify bugs that, in reality, are data dependent.

By identifying code problems without creating test cases, flow-based analysis provides developers with the "instant feedback" they need to quickly address defects and security vulnerabilities on the desktop— while they are still working on the code and it is fresh in their minds. Additionally, it prevents defects and vulnerabilities from making their way further downstream in the software development process, which is where they are much more expensive to identify and remediate.

Parasoft Static Analysis and Data Flow Analysis Technology for C and C++

Parasoft's static analysis technologies support both flow-based static analysis and pattern-based static analysis. Parasoft's flow-based static analysis technology, called Data Flow Analysis, provides effortless early detection of runtime problems and application. Parasoft Data Flow Analysis technology is available in C++test (for C and C++ code), as well as in Jtest (for Java code) and .TEST (for .NET code).

By automatically tracing and simulating execution paths through even the most complex applications—those with paths that span multiple methods, classes, and/or modules and contain dozens of sequence calls—Data Flow Analysis exposes defects that would be

[1] Books by Scott Meyers, Herb Sutter, and Andrei Alexandrescu as well as the "Gotchas" series by Dan Saks and Steve Dewhurst.

very difficult and timeconsuming to find through manual testing or inspections, and would be exponentially more costly to fix if they were not detected until runtime. Using Data Flow Analysis, developers can find, diagnose, and fix classes of software errors that can evade pattern-based static analysis and/or unit testing. Exposing these defects early in the software development lifecycle saves hours of diagnosis and potential rework.

Data Flow Analysis Rule Set

Data Flow Analysis utilizes a rule-based test configuration to apply specific tests to the source code. Rule categories built in to Data Flow Analysis include:

- **Resource Leaks –** Rules that detect potential allocation misuse of memory, pipes, file descriptors, and other system resources.
- **Memory –** Rules that detect usage of uninitialized or invalid memory.
- **Bugs –** Rules that find potential runtime errors such as division by zero, array bounding and indexing flaws, NULL pointer dereferencing, and data initialization errors.
- **Security Vulnerabilities –** Rules that detect read, write or indexing of potentially tainted data as well as potential buffer overflows.
- **Threads and Synchronization –** Rules that detect potential thread management errors.

These rules can be easily customized to map the rule logic to the organization's specific policy requirements.

Benefits of Using Data Flow Analysis

Using Data Flow Analysis, development teams gain the following key benefits:

- Perform more comprehensive testing with existing resources
- Automatically identify defects that pass through multiple classes, modules, and files
- Focus on actual defects and misuses
- Identify defects early in the software lifecycle

In The Trenches with Data Flow Analysis

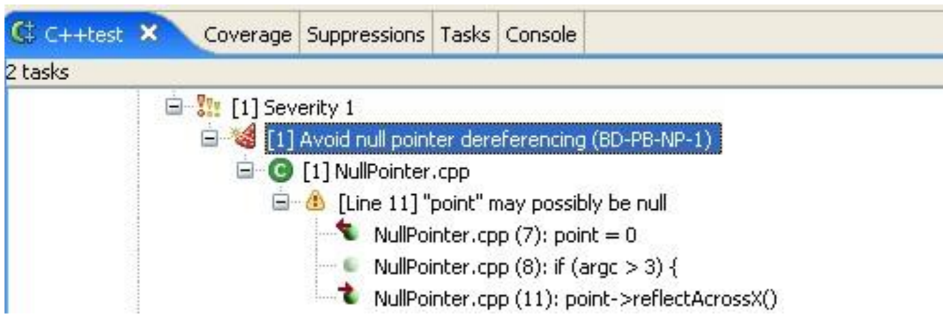
Data Flow Analysis's unique breed of static analysis determines whether application's execution paths match "suspicious behavior" profiles, which are implemented as rules. For each defect found, a hierarchical flow path details the complete execution path that leads to the identified defect, ending with the exact line of code where the bug manifests itself. To reduce the time and effort required to diagnose and correct each problem found, flow path details are supplemented with extensive annotations (for example, a null pointer dereferencing detailing where the NULL was assigned and where it may be dereferenced).

Example 1 – Dereferencing a NULL Pointer

```cpp
#include "Point.hpp"

#include <stdlib.h>

int main(int argc, char* argv[])
{
        Point* point = 0;
        if (argc > 3) {
                point = new Point(atoi(argv[1]), atoi(argv[2]));
        }
        point->reflectAcrossX();

        return 0;
}

```

Figure 1 Null pointer code example

In the above C++ code example, the dereferencing of a pointer that may be NULL at the point of execution causes a potential bug. Data Flow Analysis will find and display the code flow that caused this condition. In the C++test GUI, the violation is displayed as follows:



2 tasks
[1] Severity 1
[1] Avoid null pointer dereferencing (BD-PB-NP-1)
[1] NullPointer.cpp
[Line 11] "point" may possibly be null
NullPointer.cpp (7): point = 0
NullPointer.cpp (8): if (argc > 3) {
NullPointer.cpp (11): point->reflectAcrossX()

Figure 2 C++test Data Flow Analysis Violation Results

The violations results are displayed with each flow step that contributes to the potential bug. Line numbers in the source file are also displayed; the user may double-click each item to view and edit the source within that context.

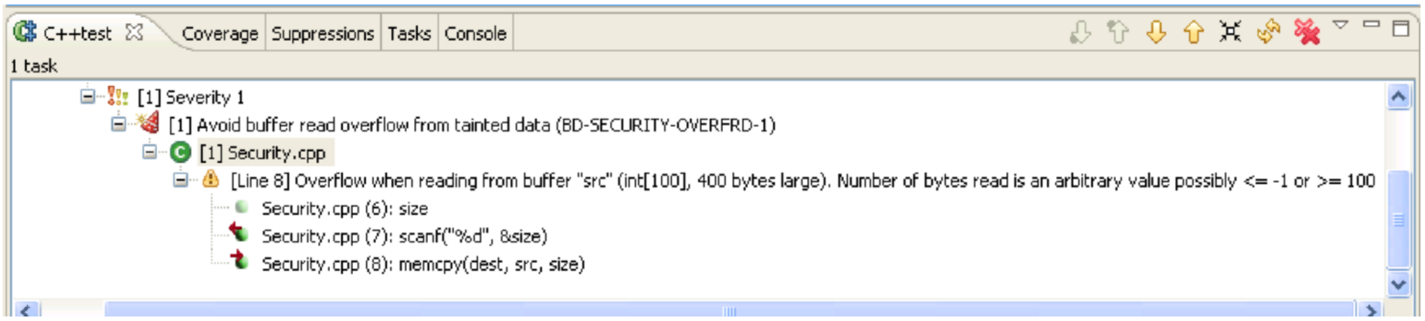
Example 2 – Security Vulnerability, Buffer Overflow

```
#include <stdio.h>
#include <string.h>

void example(int src[100], int dest[100])
{
          int size;
         scanf("%d", &size);
    memcpy(dest, src, size); // VIOLATION ("size" is an arbitrary value possibly < 0 or > 100)
}
```

Figure 3 - Buffer Overflow Security example

In Figure 3 above, a potential buffer overflow exists in the code sample. The possibility of buffer overflow is a severe security threat. If an application has a vulnerability of this kind, it can be exploited to execute arbitrary code and gain full control over the application. This is especially true with local arrays allocated on the processor stack.

When C++test is run with Data Flow Analysis and the rule “BD-SECURITY-OVERFRD-1” enabled, the following violation is reported to the developer:



C++test | Coverage | Suppressions | Tasks | Console

1 task

```
[1] Severity 1
  [1] Avoid buffer read overflow from tainted data (BD-SECURITY-OVERFRD-1)
    [1] Security.cpp
      [Line 8] Overflow when reading from buffer "src" (int[100], 400 bytes large). Number of bytes read is an arbitrary value possibly <= -1 or >= 100
        Security.cpp (6): size
        Security.cpp (7): scanf("%d", &size)
        Security.cpp (8): memcpy(dest, src, size)
```

Figure 4 - C++test Security Violation Results

Data Flow Analysis finds a use of memcpy with an untested and arbitrary size limit. When corrected in the **Figure 5** sample that follows, the size parameter is first tested for range validity before the data is copied.

```
#include <stdio.h>
#include <string.h>

void example(int src[100], int dest[100])
{
        int size;
        scanf("%d", &size);
        if (size >= 0 && size <= 100) {
                memcpy(dest, src, size); // NO VIOLATION
        }
}
```

Figure 5 - Corrected Security Violation

Learning More

Parasoft Data Flow Analysis is available with Parasoft Jtest, C++test, and .TEST. To learn more about Parasoft Data Flow Analysis, contact Parasoft as described below, or visit http://www.parasoft.com.

About Parasoft

For 25 years, Parasoft has researched and developed software solutions that help organizations deliver defect-free software efficiently. By integrating Development Testing, API/cloud/SOA/composite app testing, and service virtualization, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing with requirements traceability, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (0)1923 858005
Germany: Tel: +49 89 4613323-0
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Other Locations

See http://www.parasoft.com/contacts