Static Code Analysis for C++ Applications

C++ Core Meet up 28-12-2017

29.12.2017

Agenda:

- Short Introduction, who am I, from where?
- What is Static Code Analysis?
- Pattern matching SCA vs Data Flow based SCA
- Short demo of Analysis, patteran matching and Data Flow

אי. אס. אל. מערכות תוכנה בע"מ

מרכז מומחיות לכל האספקטים של ניתוח קוד סטאטי ובדיקות דינאמיות – שרות מלא מקצה לקצה

- **יעוץ מקצועי, התקנה והדרכה קורסים.**
- **אינטגרציה לתוך מערך הבדיקות והפיתוח בארגון**
- **פיתוח מערכי בדיקות ובדיקות ייעודיות**
- יעוץ בכתיבת קוד נכון ותיקון בפועל (או יעוץ לדרכי תיקון מעשיות) של טעויות שהתגלו על ידי כלי הבדיקה.
- הוקמה ב 2005 ממוקמת ברמת גן , ומעסיקה 9 עובדים.
- מייצגת את חברת Parasoft INC מאז שנת 2007
- 170 לקוחות פאראסופט בישראל
- לקוחות עיקריים – חברות ציוד רפואי (Philips and GE Medical , Covidien), פיננסים (בורסת ת"א, בנק דיסקונט), התעשייה הביטחונית (תעשייה אווירית, אלביט) , מגזר השבבים (Broadcom , Marvell) ;

29.12.2017

Parasoft Company Background

- Founded in 1987, privately held
- Founder and CEO until 2012 Dr. Adam Kolawa from CalTech
- Headquarters in Monrovia, CA
- 22 locations and 500+ employees worldwide
- 80 million LOC VS 290 Developers
- Analyst Technical Innovator
- 53 US patents for software technology
- 17,000+ customers worldwide
- 85% Fortune 100 Companies

Parasoft – Market Postion

Figure 4 Forrester Wave™: Modern Application Functional Test Automation, Q2 '15

Risky Bets
Contenders
Strong Performers
Leaders

Strong
Current offering
Weak

Parasoft
IBM
Borland
Tricentis
CA Technologies
HP
Microsoft
SmartBear
TestPlant

Market presence

The Forrester Wave™
Smart data for smart decisions

Go to Forrester.com to download the Forrester Wave tool for more detailed product evaluations, feature comparisons, and customizable rankings.

Parasoft Automated Tools - Overview

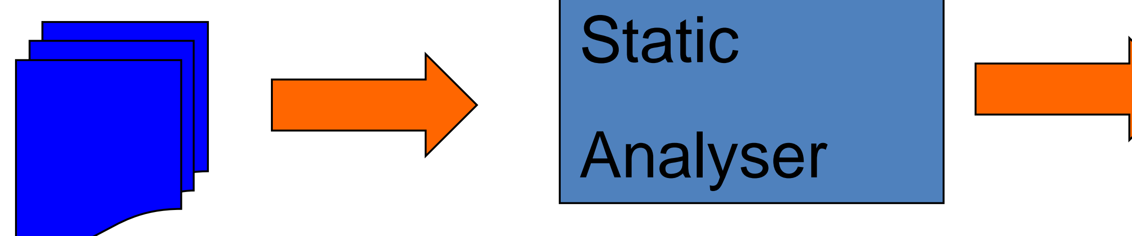
Tool	Features	
PARASOFT C++test™	• **Coding Standards enforcement** • **Automatic Unit testing** • **Embedded Support** • Security Testing	• Data Flow analysis • **Auto Stub generation** • Regression Test • Code /Test Coverage
PARASOFT .TEST™	• Coding Standards enforcement • Automatic Unit testing • Realistic auto gen functional tests • Regression Testing	• Data Flow analysis • Security Testing • Auto Stub generation • Test Coverage
PARASOFT Jtest®	• Coding Standard Enforcement • Automatic Unit testing • Realistic auto gen functional test • Regression test	• Data Flow analysis • Security Testing • Auto Stub generation • Test Coverage
PARASOFT Insure++®	Automated Runtime Memory Defect Detection for C/C++	
PARASOFT SOAtest™	• End to End Testing • Web Application Testing • Functional/Integration Testing • Application Behavior Virtualization	• Security Testing • Regression Testing • Policy Enforcement • Load Testing
PARASOFT Virtualize	• Application Behavior Virtualization • Development/test environment management	
Mobile Test	Ensures the security reliability and performance of enterprise-grade mobile applications.	
PARASOFT Concerto™ A Parasoft AEP Technology™	Decision Support Mechanism- Visibility, Control and Management of SDLC	

What is it “Static Code Anlysis” and what is it used for?

Static program analysis is the analysis of computer software that is performed without actually executing programs.

In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. ...

Static analyzers: General form



Document
Eg. Source code: .C.CPP .H .CC ,CS etc..

- Syntax violation
- Coding Standards Deviation
- Data flow info
- Control flow info
- Defects
- Errors
- Bugs

Parasoft Proprietary and Confidential

Static Code Analysis is an Automatic code review tool!

Usually performed during coding (recommended) or *after* the coding finished (after compilation, after integration build)

Serves same goals as code review

- **Excellent for enforcing compliance to standards**
- **Helps to eliminate *certain* bugs**
- **Helps to identify *certain* design/implementation flaws**
- **Provides *certain* educational value**

In simple words…..

- It is an advanced and easy to use “debug tool”
- It is a Coding Standards Enforcer
-
- It is a programmers training tool
- It is the only viable way to expose Data Flow related coding defects
- It is a modern and automated way of doing what we always did manually for finding bugs and ensuring source code quality…..

How we did things in the past and how today

Impact (benefit) of Static Anlysis is high

140000
120000
100000
80000
60000
40000
20000
0

Defects

Without Static Analysis
With Static Analysis

1M 2M 3M 4M 5M 6M 7M 9M 10M

Lines of code

Static Analysis may reduce defects by a factor of 6!

Source: Capers Jones, Software Productivity Group, Inc.

Parasoft Test: 3x Static Analysis

- ***Pattern-Based Static Analysis***
  - Increases productivity by preventing errors
  - Extensive breadth of rules
    - 2300 for C/ C++
    - Over 1,000 for Java
    - Over 700 for .NET
  - Parasoft Test rule quality based on over 20 years of research
    - **No false positives / No False Negative**
    - Depth of analysis
  - Graphical interface for custom rule creation and customization
  - Extensive security Rulesets for (PCI, OWASP, Sun Java Security…
- ***Flow-Based Static Analysis***
  - Finds bugs
  - Deep, multi-file path analysis
  - **Very low false positives**
- ***Metrics Analysis***
  - Finds complex code prone to errors
  - Directly pinpoints areas of code/application prone to errors
  - Large breadth of metrics available

Pattern Matching issue

- **Prefer lambdas over std::bind, std::bind1st and std::bind2nd**
  [CODSTA-MCPP-07-2]
- Scott Meyers, "Effective Modern C++, 42 specific ways to improve your use of C++11 and C++14", O'Reilly Media, Inc., Copyright 2015, Chapter 6: "Lambda Expressions", Item 34: "Prefer lambdas to std::bind“

(Since C++17, std::bind1st and std::bind2nd are removed from the Standard)

Why?

This rule detects when 'std::bind', 'std::bind1st' or 'std::bind2nd' are used in code.

Older versions of the Standard used 'std::bind', 'std::bind1st' or 'std::bind2nd'. C++11 allows you to use lambda expressions that are more readable, more expressive and make your code easier to optimize. In C++11, lambda expressions cannot replace polymorphic function objects and they do not offer move capture. However, C++14 introduces polymorphic lambda expressions, as well as generalized lambda capture, which enables you to replace 'bind' in all cases.

Pattern Matching issue

EXAMPLE

```
#include <functional>

int f_a(int a, int b);

template <typename T> void ft(T t)
{
    int a;
    t(a);
}

void foo( void )
{
    auto fn = std::bind(f_a, 10, std::placeholders::_1);     // Violation

    int a = 10;
    ft(std::bind(f_a, a, std::placeholders::_1));            // Violation
}
```

EXCEPTIONS The rule does not report a violation when a lambda is passed to 'std::bind' as an argument. This may happen when move capture

Exceptions to the rule?

PARASOFT

EXCEPTIONS

The rule does not report a violation when a lambda is passed to 'std::bind' as an argument. This may happen when move capture is not available(in C++11).
For example:

```
  std::bind([] (int a) {}, 10);
```

Rules Wizard

Translation Unit | xor

Body / File | and

IsInOutputScope / T

Body / a(b) | and

Left Hand Side / Global Function | and

Name / ^(bind|bind1st|bind2nd)$

1 | A | and

Count / $$>0

Filename / ^(functional|functional\.h|xfunctional|
xfunctional\.h|_function|_function\.h|
stl_function|stl_function\.h|
functional_iterate|functional_iterate\.h|
binders|binders\.h|xxbind1|xxbind1\.h)$

nand

Arguments / Argument | and

Type / Class | and

IsLambdaClosureClass / T

TemplateInstances / Functions | and

Body / a(b) | and

Left Hand Side / Global Function | and

Name / ^(bind|bind1st|bind2nd)$

1 | B | and

Count / $$>0

Filename / ^(functional|functional\.h|xfunctional|
xfunctional\.h|_function|_function\.h|
stl_function|stl_function\.h|
functional_iterate|functional_iterate\.h|
binders|binders\.h|xxbind1|xxbind1\.h)$

nand

Arguments / Argument | and

Type / Class | and

IsLambdaClosureClass / T

Pattern Matching issue

REPAIR

```
#include <functional>

int f_a(int a, int b);

template <typename T> void ft(T t)
{
    int a;
    t(a);
}

void foo( void )
{
    auto l_f1 = [](int a){ return f_a(10, a); };     // OK

    int a = 10;
    ft([a](int b){ return f_a(a, b); });              // OK
}
```

- There is One thing the Code Review hardly can do....
- Inter procedural Crash Causing Defects

the Solution:

Data Flow Analysis

What Can be found with Data Flow Analysis ?

PARASOFT

- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- passing large parameters by value
- Under allocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes

C++test – Bug Detective

Data Flow Analysis

How does it work?

3d Generation SCA tools – 2006 – to present,

source code

```
int a, b;
a = 2;
b = a*2 + 1;
```

target code

```
SET   R1,2
STORE #0,R1
SHIFT R1,1
STORE #1,R1
ADD   R1,1
STORE #2,R1
```

Compiler components

Character Stream

Lexical Analyzer

Token Stream

Syntax Analyzer

Syntax Tree

Semantic Analyzer

Decorated Syntax Tree

Intermediate Code Generator

Intermediate Representation

Machine-Independent Code Optimization

Intermediate Representation

Code Generator

Target Machine Code

Machine-Dependent Code Optimization

Target Machine Code

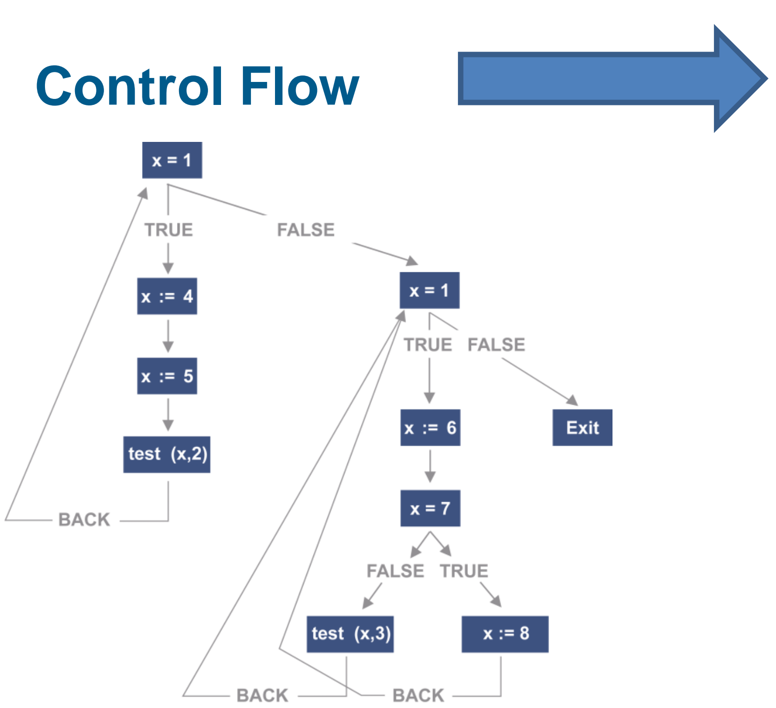
Software DNA Map

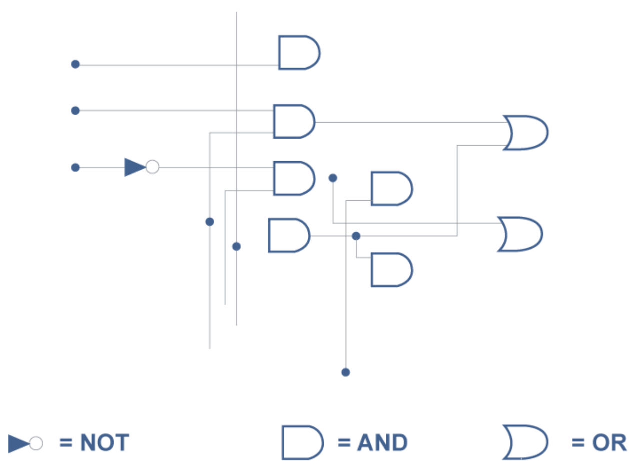
- An accurate representation of a software system based on understanding all operations that the build system performs as well as an authentic compilation of every source file in that build system.
- Software DNA Map enables static code analysis to overcome its previous limitations of excessive false positives and deliver accurate results that developers can put to immediate use.

Comprehensive: Bit-Accurate

- Bit-accurate representation of the data and logic of the software system allows SAT solvers to explore all possible values
- Enables integer overflow detection and optimal false path pruning

Control Flow → **Bit-Accurate Representation**

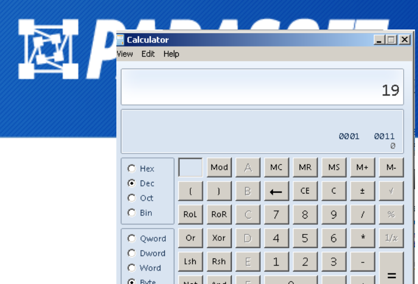




Example of a Control Flow Graph

```
1. d:=0;
2. while (x<y) {
3.    x:=x+3;
4.    if (x+y < 100)
5.       s:=s+x+y;
6.    else
7.       s:=s+x-y;
8. }
```

Boolean Satisfiability (SAT Solver) using the DNA map



- Take the expression A==19 (A is a 8 bit char) ,
- DNA mapping will convert it to :

! a 7 ^ ! a 6 ^ ! a 5 ^ a 4 ^ ! a 3 ^ ! a 2 ^ a 1 ^ a 0

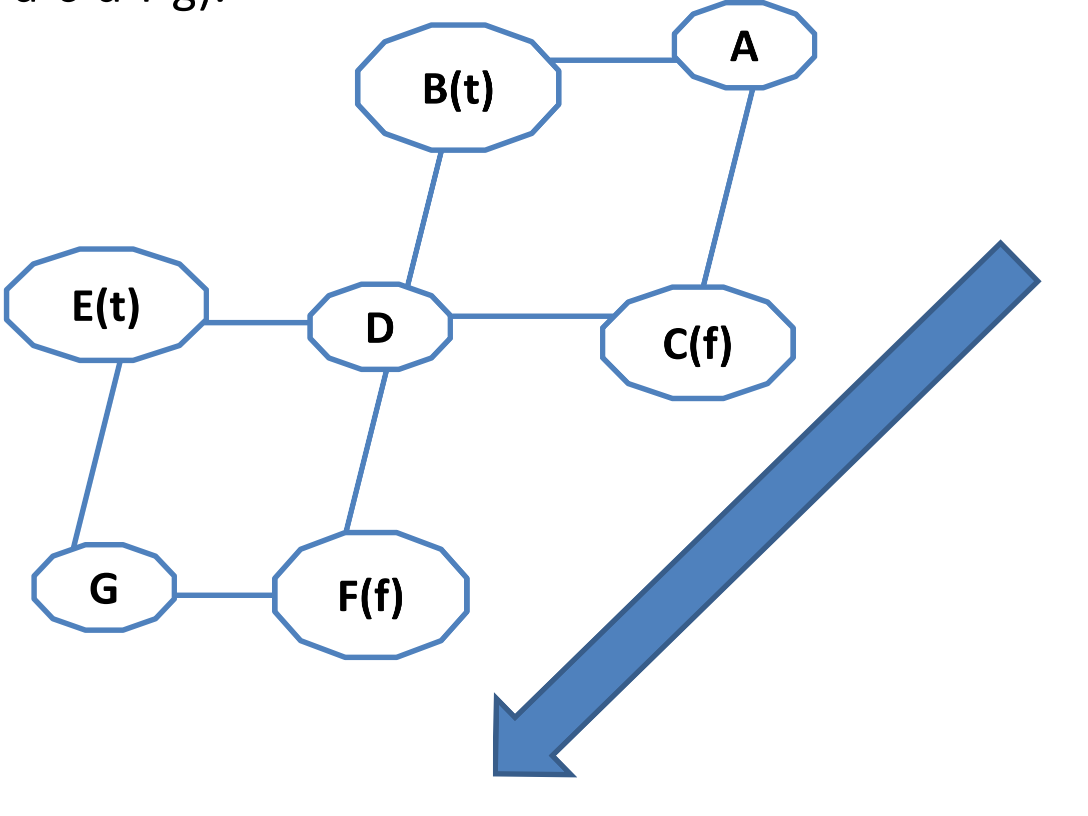
(a7 is the high bit)

- Plugging this into a SAT Solver would render the following assignment of variables for the formula to be satisfied:
- **a 0 = True . (1) a 1 = True . (1) a 2 = False (0). a 3 = False (0). a 4 = True (1). a 5 = False (0). a 6 = False (0) a7 = False (0)**
- **We got** 00010011 =19

- Once the entire Software DNA Map is represented in this format of TRUES, FALSES, NOTS, ANDS, and
- ORS, a wide variety of formulas can be constructed from this representation and SAT solvers can be applied to analyze the code for additional, more sophisticated quality and security problems. It is this bit-accurate representation of the software that enables more precise static analysis than previously was possible based solely on path simulation.

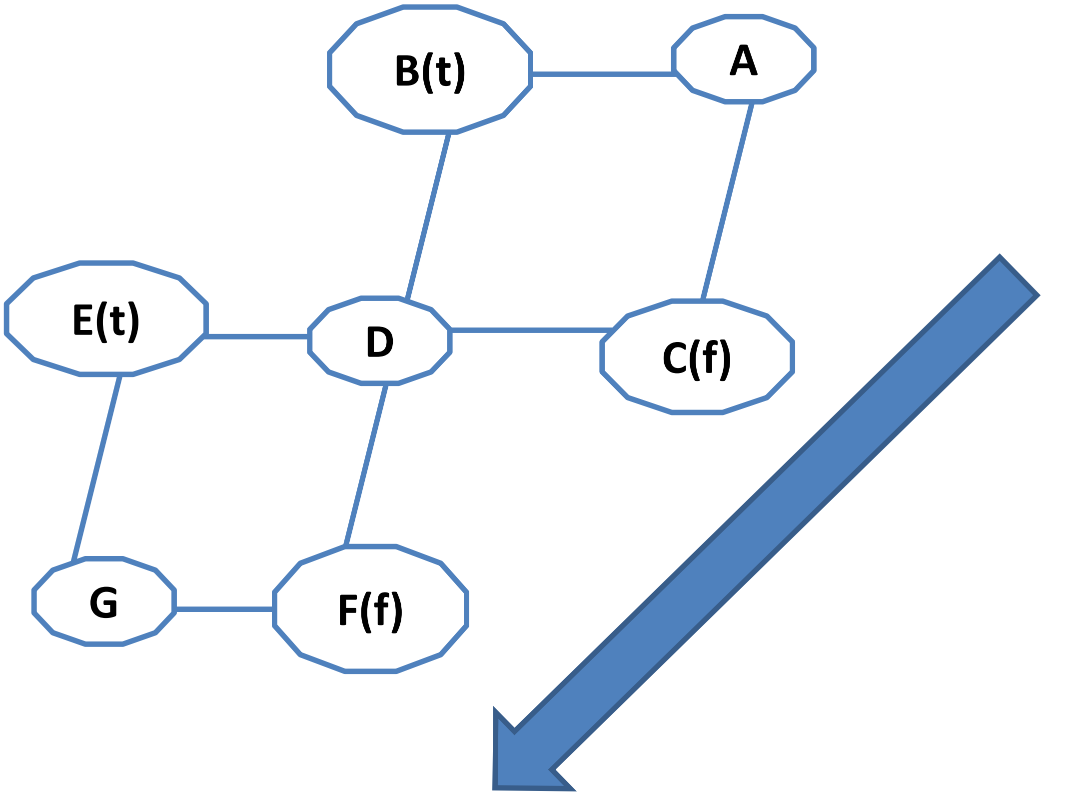
Path Simulation

- There are clearly four paths through this code base (a-b-d-e-g, a-c-d-e-g, a-b-d-f-g, a-c-d-f-g).



Path Simulation , enter the SAT solver

- Let’s assume we have the following expressions
- [a] : i f (x = = 0) [d] : i f (x ! = 0)



The SAT Solver

- The SAT solver see “x = = 0   AND   x ! = 0”
- The SAT solver says “this cannot be satisfied boolianly”
- while there might appear to be 4 paths through the control flow graph, we know that because of the dependency between the condition of (a) and condition of (d), there are only 2 paths through the code base.
- If the analysis decides to explore the path **a-b-d-e-g,** this would be The SAT solver see “x = = 0   AND   x ! = 0”
- The SAT solver says “this cannot be satisfied boolianly”
- while there might appear to be 4 paths through the control flow graph, we know that because of the dependency between the condition of (a) and condition of (d), there are only 2 paths through the code base. If the analysis decides to explore the path a-b-d-e-g, this would be a FALSE path because it’s impossible to execute at runtime. Moreover, if the analysis reported a defect on this path, that defect would clearly be a false positive since that path cannot exist when running the program.

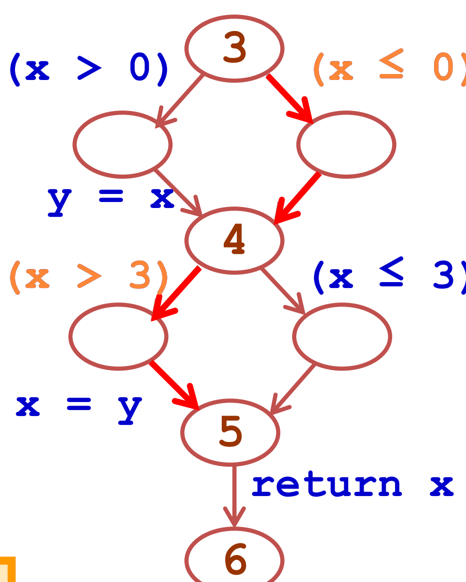
- a FALSE path because it’s impossible to execute at runtime. Moreover, if the analysis reported a defect on this path, that defect would clearly be a false positive since that path cannot exist when running the program.

False Positive Problem: False Errors

- ***false error**: reported by analyzer but not in fact a latent error in program*

```
1   int f(int x) {
2    int y;
3    if (x > 0) y = x;
4    if (x > 3) x = y;
5    return x;
6   }
```

Warning 644:
Variable 'y' (line 2) may not have been initialized



10-Mar-05

f.h
```
1  int f(int x) {
2   int y;
3  if (x > 0) y = x;
4  else;
5  y++;
6   return x;
7  }
```

f.c
```
1  #include <f.h>
2   void main (void)
3   f(1); //No Violation
4    Do something…...
5        f(-1);//Violation
6  }
```

Typical Defect….

```
void buffer_size_example()
{
char dest[128]; char source[256]; strncpy(dest,
   source, strlen(source));
}
```

// This will flag an error as the size argument to strncpy() // can possibly be up to 255, yet the destination only has // room for 128 elements (127 chars and the null termination).

But it is never that obvious….

Buffer overrun

Or even looking remotely like that….

```
void func (char *passedStr)
    {
char localStr[4];
strcpy(localStr, passedStr); // length of passedStr is not
   checked
   }

int main (int argc, char **argv)
    {
func(argv[1]);
          }
```

It can look like that….. History in the making
The code that made the iPhone what it is…

The *LIBTIFF VULNERABILITY*

```
static int
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{
switch (dir->tdir_type) {
case TIFF_BYTE:
case TIFF_SBYTE:
{
uint8 v[4];
return TIFFFetchByteArray(tif, dir, v)
&& TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
        }
        case TIFF_SHORT:
        case TIFF_SSHORT:
                {
        uint16 v[2];
        return TIFFFetchShortArray(tif, dir, v)
        && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
                }
        default:
        return 0;
                }
                        }
```

Live Demo!

Sensor - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP PARASOFT

Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'Sensor' (1 project)
  - Sensor
    - External Dependencies
    - Header Files
      - Source.h
    - Resource Files
    - Source Files
      - sensor.c
    - tests

Test Configurations

Test Configurations:

- dotTEST
  - User-defined
  - Builtin
  - Team
- C++test
  - User-defined
    - Application Monitoring
    - Static Analysis
      - CATH_1_FDA C++ P
      - Dani_1
      - ISO 26262 Recomme
      - My_BugDetective Ag
      - My_Parasoft's Recon
      - OWASP Top 10 Sec
      - Parasoft's Recommer
      - Parasoft's Recommer
      - XYZ-1
      - ywuyyquyquy
    - Unit Testing
    - Example Configuration
    - New Configuration
  - Builtin
  - Team

Name: CATH_1_FDA C++ Phase 2

Parent: Disconnect

Scope | Static | Generation | Execution | Code Review | Common | Goals

- [x] Enable Static Analysis
- [x] Limit maximum number of tasks reported per rule to: 1000
- [ ] Do not apply suppressions
- [ ] Analyze files with parse errors

Rules Tree | Metrics | BugDetective Options

Number of rules: 1,868 total; 149 enabled; 0 hidden

- BugDetective (License Required) [BD] - (16/26 enabled)
  - Miscellaneous [BD-MISC] - (0/1 enabled)
    - [ ] Always close transactions [BD-MISC-TRANS - 1]
  - Possible Bugs [BD-PB] - (10/11 enabled)
    - [x] Avoid accessing arrays out of bounds [BD-PB-ARRAY - 1]
    - [x] Avoid use before initialization [BD-PB-NOTINIT - 1]
    - [x] Avoid null pointer dereferencing [BD-PB-NP - 1]
    - [x] Avoid buffer overflow due to defining incorrect format limits [BD-PB-
    - [x] Avoid overflow due to reading a not zero terminated string [BD-PB-
    - [x] Avoid overflow when reading from a buffer [BD-PB-OVERFRD - 1]
    - [x] Avoid overflow when writing to a buffer [BD-PB-OVERFWR - 1]
    - [x] Avoid division by zero [BD-PB-ZERO - 1]
    - [ ] Avoid conditions that always evaluate to the same value [BD-PB-C
    - [x] Do not check for null after dereferencing [BD-PB-DEREF - 2]

Hide Disabled | Show All | New | Import | Reload | Edit Rulemap | Printable Docs

New | Delete | Revert | Apply | Run Test | Close

Output — Show output from:

Error List | Quality Tasks | Test Progress | Coverage | Output | Find Results 1

eady — Ln 9 Col 16 Ch 16

תודה על ההקשבה

דני לייזרוביץ'