DEFENCE R&D DÉFENSE

So You Have to Verify Software?

A Quick Look at What You Should Expect

Frédéric Michaud & Frédéric Painchaud
Defence Scientists

R et D pour la défense Canada

Defence R&D Canada

Canada

Introduction

- Team's past projects in software security
  - MaliCOTS: malicious code detection (97-01)
  - SOCLe: UML design verifier (02-05)
  - Secure design patterns catalogue (03-04)
  - ***Verification tools evaluation (05-06)***
    - 27 tools
    - 25 segments of code containing known defects spread into a pre-screened application code
    - For detailed results, a complete report is available
      - No raw results

Focus of this Presentation

- Fully automatic source code verification tools
- C and C++ programming languages
- Bugs related to C/C++ use
  - Problems with impacts on security
- Practical considerations

Plan

1. Getting started
2. Choosing the right tool for the job
3. Running the verification process
4. Analyzing results
5. Conclusion

1. Getting Started

- What kind of bugs can be detected?
- What kind of applications can be verified?
- Who should do the verification?
- When should the verification take place?

Some Terminology

- Error → Execution
  - Event that occurs when the behaviour of a program diverges from “what it should be”
- Defect → Code
  - Cause of an error, a set of program instructions
  - Can be the absence of something
    - Data validation
- Vulnerability → Exploitation
  - Defect allowing a user to control the program execution when it should not

C/C++ Defects - Some Observations

- Found bugs impacting software quality
  - Design problems (~40%)
  - Programming problems (~60%)
- Cause of programming problems
  - Mistakes and ignorance
  - Badly designed programming languages
    - Too easy to create defects
    - Defects with dangerous consequences

A programming language is low level when its programs require attention to the irrelevant. – Epigram 8, Alan J. Perlis, Epigrams on Programming, SIGPLAN Notices, Vol. 17, No. 9, September 1982, pp. 7-13.

C/C++ Defects - Some Observations

- Many defects are not always activated
  - They will not always generate errors
  - Complex conditions have to be met
  - Input values play an important role
- Most defects are composite
  - Cannot be attributed to only one program instruction
  - In one function, in one component, in many components, in different technologies…
- Simply looking for “bad” instructions won’t get you far…

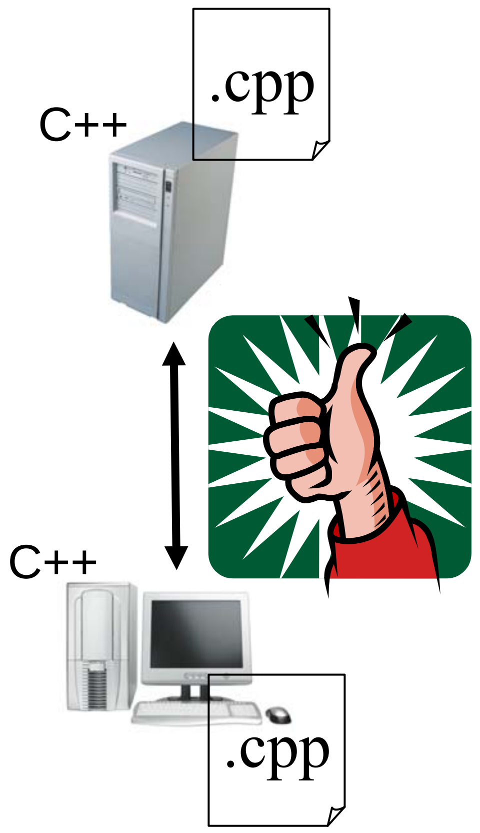
Bugs Typically Found

- Low-level problems
- Memory management faults
  - Using freed memory
  - Underallocated memory for a given type
- Overrun and underrun faults
  - Buffer overflows
  - Dereferencing past-the-end C++ iterator
- Pointer faults
  - Return of a pointer to a local variable
  - Dereferencing a null pointer

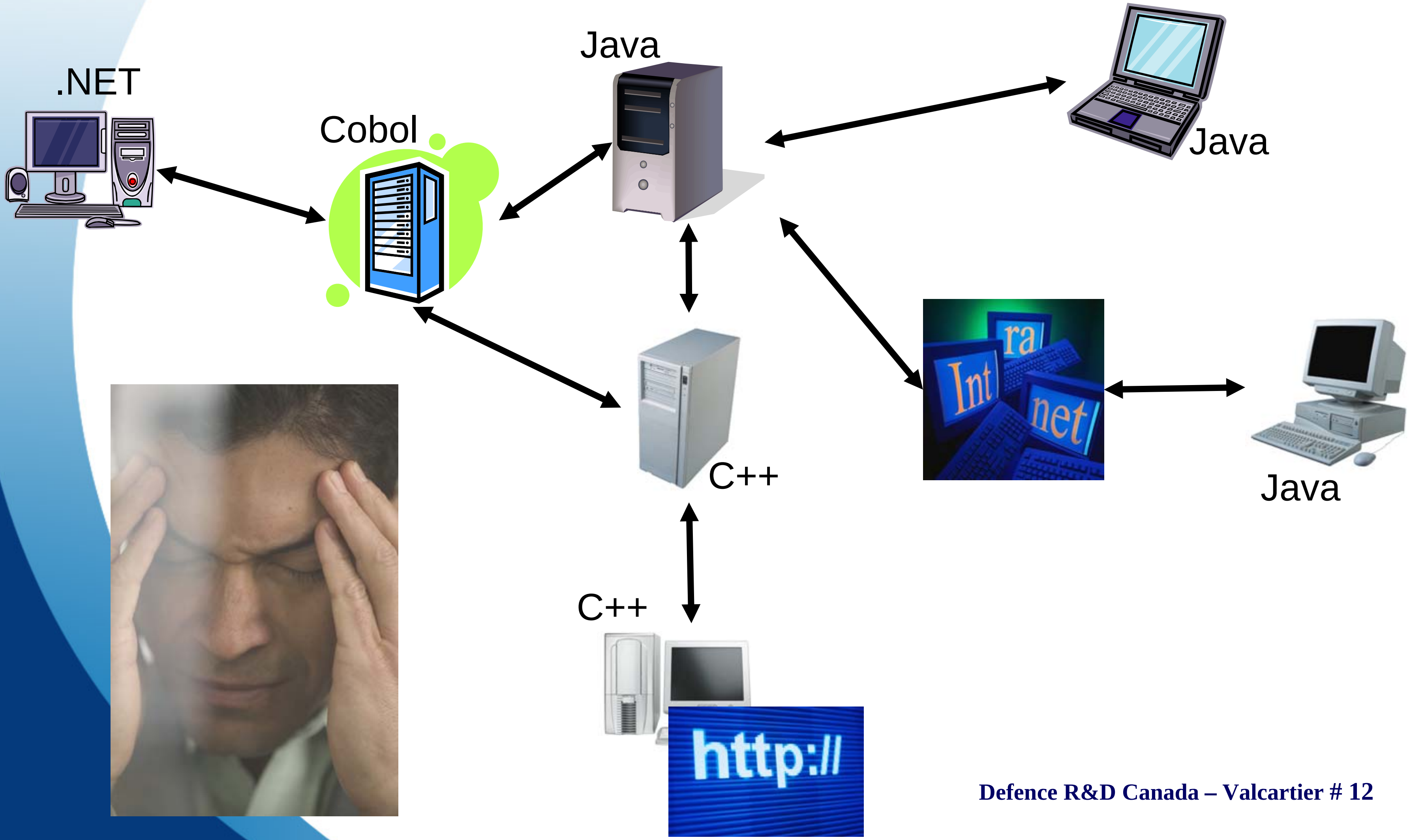
Problems Still Out of Reach

- Anything related to design and security
  - Malicious code in general (no signature)
  - Backdoors in access control
  - Covert channels and illicit communication
  - Key logging, audio recording
  - Encryption keys generation & management
- However
  - Many vulnerabilities are caused by low-level problems
    - Buffer overflows, memory leaks, etc.

Using Verification Tools on Simple, Homogeneous Systems



Using Verification Tools on Complex, Heterogeneous Systems



Using Verification Tools on Your Own Code or on Others'

- Quality Assurance Team or Customer
  - Not so familiar with the code
  - Diagnostic takes longer to analyze
  - Change impact analysis takes longer
- Programmer
  - Knows the code
  - Quickly understands the diagnostic
  - Analyzes the change impact more rapidly
  - Learns from past mistakes
  - Less hurt feelings
- This does not break the principle that QA should be done independently of design and implementation

Using Verification Tools During or After Development

- After Development
  - Too many problems in one shot
  - Hard to prioritize
  - Changes can cause more problems
  - Long iterative process
- During Development
  - Few problems, corrected right away
  - Localized problems, few interactions
  - Code is still mastered by programmer

2. Choosing the Right Tool for the Job

- What kind of tools are available?
- What are the important tool attributes?
- What to expect about these attributes?
- Which tools do you recommend?

Automatic Source Code Analyzers

- Static Analyzers
  - Detect defects by compiling (without executing) the program
  - Formal methods and advanced heuristics
  - Sophisticated and expensive tools
- Dynamic Analyzers
  - Detect errors by running the program
  - Small checks are added to the code
  - Simple to use and affordable tools

Important Tool Attributes

- Accuracy
  - False positives can slow the review a lot
  - Static: 30% false positives
  - Dynamic: 0% false positives (in theory)
- Requirements on the program to be verified
  - Can make the verification impossible
  - Static: should compile, partial verification possible, needs source code
  - Dynamic: should compile, link and execute

Important Tool Attributes

- Scalability
  - Some tools won’t work with large programs
  - Static: max from 20 KLOC to **over** 100 KLOC
  - Dynamic: no limit
- Speed
  - Regular verification (nightly builds) can be impractical
  - Static: from 15 minutes to 15 hours for 10 KLOC
  - Dynamic: verification time = test cases * execution time

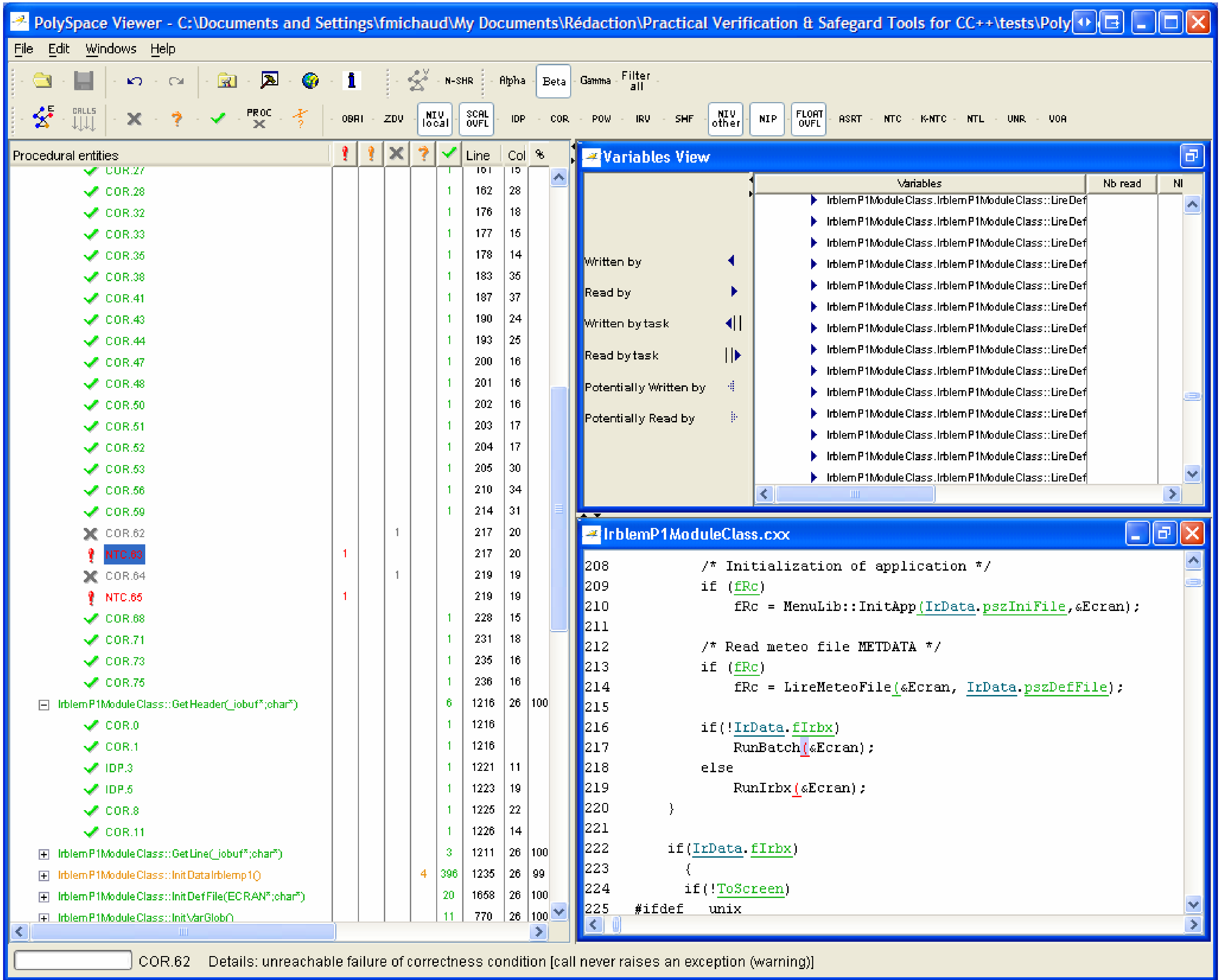
KLOC: thousands of lines of code

Important Tool Attributes

- Coverage
  - Some problems can be missed
  - Static: covers all (or most) paths (by means of abstraction), but not much value domains (they are abstracted)
  - Dynamic: one execution only, needs many tests
- Versatility
  - Some tools look for many kinds of problems
  - Static: from focused to very general
  - Dynamic: mostly memory management

Important Tool Attributes

- Diagnostic
  - A good diagnostic can speed reviews a lot
  - Static: from silly to detailed execution trace with hints
  - Dynamic: very debugger-like
- Documentation
  - Description of problems, impacts, and corrections
  - Static: generally much more complete
  - Dynamic: a limited number of problems



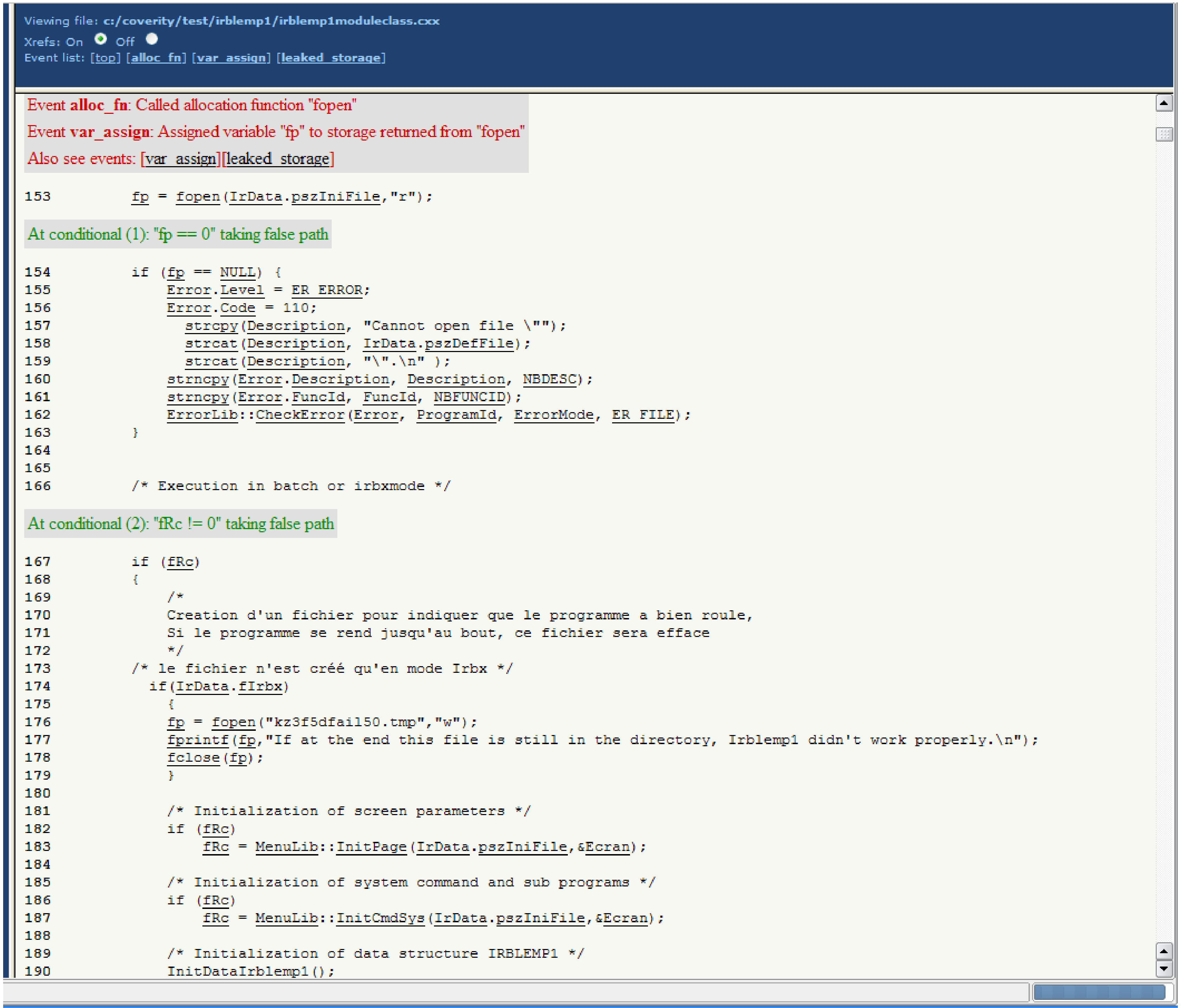
Procedural entities	!	!	✗	?	✓	Line	Col	%
COR.27					1	161	15	
COR.28					1	162	28	
COR.32					1	176	18	
COR.33					1	177	15	
COR.35					1	178	14	
COR.38					1	183	35	
COR.41					1	187	37	
COR.43					1	190	24	
COR.44					1	193	25	
COR.47					1	200	16	
COR.48					1	201	16	
COR.50					1	202	16	
COR.51					1	203	17	
COR.52					1	204	17	
COR.53					1	205	30	
COR.56					1	210	34	
COR.59					1	214	31	
COR.62			1			217	20	
NTC.63	1					217	20	
COR.64			1			219	19	
NTC.65	1					219	19	
COR.68					1	228	15	
COR.71					1	231	18	
COR.73					1	235	16	
COR.75					1	236	16	
IrblemP1ModuleClass::GetHeader(_iobuf*,char*)					6	1216	26	100
COR.0					1	1216		
COR.1					1	1216		
IDP.3					1	1221	11	
IDP.5					1	1223	19	
COR.8					1	1225	22	
COR.11					1	1226	14	
IrblemP1ModuleClass::GetLine(_iobuf*,char*)					3	1211	26	100
IrblemP1ModuleClass::InitDataIrblemp1()				4	396	1235	26	99
IrblemP1ModuleClass::InitDefFile(ECRAN*,char*)					20	1658	26	100
IrblemP1ModuleClass::InitVarGlob()					11	770	26	100

Written by
Read by
Written by task
Read by task
Potentially Written by
Potentially Read by

Variables	Nb read	Nl
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		
IrblemP1ModuleClass.IrblemP1ModuleClass::LireDef		

```
208            /* Initialization of application */
209            if (fRc)
210                fRc = MenuLib::InitApp(IrData.pszIniFile,&Ecran);
211
212            /* Read meteo file METDATA */
213            if (fRc)
214                fRc = LireMeteoFile(&Ecran, IrData.pszDefFile);
215
216            if(!IrData.fIrbx)
217                RunBatch(&Ecran);
218            else
219                RunIrbx(&Ecran);
220        }
221
222        if(IrData.fIrbx)
223          {
224          if(!ToScreen)
225    #ifdef   unix
```

COR.62   Details: unreachable failure of correctness condition [call never raises an exception (warning)]



Viewing file: c:/coverity/test/irblemp1/irblemp1moduleclass.cxx
Xrefs: On ● Off ○
Event list: [top] [alloc_fn] [var_assign] [leaked_storage]

Event **alloc_fn**: Called allocation function "fopen"

Event **var_assign**: Assigned variable "fp" to storage returned from "fopen"

Also see events: [var_assign][leaked_storage]

```
        fp = fopen(IrData.pszIniFile,"r");
```

At conditional (1): "fp == 0" taking false path

```
        if (fp == NULL) {
            Error.Level = ER_ERROR;
            Error.Code = 110;
              strcpy(Description, "Cannot open file \"");
              strcat(Description, IrData.pszDefFile);
              strcat(Description, "\".\n" );
            strncpy(Error.Description, Description, NBDESC);
            strncpy(Error.FuncId, FuncId, NBFUNCID);
            ErrorLib::CheckError(Error, ProgramId, ErrorMode, ER_FILE);
        }


        /* Execution in batch or irbxmode */
```

At conditional (2): "fRc != 0" taking false path

```
        if (fRc)
        {
            /*
            Creation d'un fichier pour indiquer que le programme a bien roule,
            Si le programme se rend jusqu'au bout, ce fichier sera efface
            */
        /* le fichier n'est créé qu'en mode Irbx */
          if(IrData.fIrbx)
            {
            fp = fopen("kz3f5dfail50.tmp","w");
            fprintf(fp,"If at the end this file is still in the directory, Irblemp1 didn't work properly.\n");
            fclose(fp);
            }

            /* Initialization of screen parameters */
            if (fRc)
                fRc = MenuLib::InitPage(IrData.pszIniFile,&Ecran);

            /* Initialization of system command and sub programs */
            if (fRc)
                fRc = MenuLib::InitCmdSys(IrData.pszIniFile,&Ecran);

            /* Initialization of data structure IRBLEMP1 */
            InitDataIrblemp1();
```

SV.BRM.VARIABLE

Access problem - report flags Windows system calls that use a variable as the hkey parameter. Use of a variable allows for possible manipulation of the variable's value and arbitrary access to registry values.

This error type is produced for eight functions which should not use registry key variables that can be modified by any logged in user.

- RegConnectRegistry
- RegCreateKey
- RegCreateKeyEx
- RegLoadKey

- RegOpenKey
- RegOpenKeyEx
- SHRegCreateUSKey
- SHRegOpenUSKey

Defect Attributes

Name	Value
Defect Code	SV.BRM.VARIABLE
Category	Security Vulnerabilities/ Access Problems/ Least Privilege
Title	Use of Variable as 'hkey' in Registry Manipulation Function
Message	Function '{0}' must use a constant value for parameter {1}. Use of a variable allows for possible manipulation of the value and arbitrary access to registry values.
Enabled (default)	false
Severity (default)	Severe (2)
Applicable language	C/C++
Customizable	false

Vulnerability and risk

Using user-accessible variables for registry manipulation can lead to arbitrary access and possible modification of registry keys and values.

Mitigation and prevention

Use explicit constants as hkey parameters for registry manipulation functions. Review the use of registry by the application.

Example

```
1    // SV.BRM.VARIABLE
2    void foo(LPTSTR lpMachineName, HKEY hkey, PHKEY phkResult) {
3        RegConnectRegistry(lpMachineName, hkey, phkResult);
4    }
```

Klocwork products produce a defect report like the following:

test.c:2:Severe:Function 'RegConnectRegistry' must use a constant value for parameter 2. Use of a variable allows for possible manipulation of the value and arbitrary access to registry values.

Recommended Tools

- I am NOT saying that other products are not worth buying…
- Very rapidly evolving

Recommended Tools

- Coverity Prevent, KlocWork K7 (Static)
    - Whole applications compiled with makefiles, VStudio projects
- PolySpace for C++ (Static)
    - Small sections of critical code in which runtime exceptions should never happen
- Parasoft Insure++ (Dynamic)
    - Testing hybrid systems based on many heterogeneous components
    - Integrated into test cases

3. Running the Verification Process

- Integration issues
  - Project files used by major IDEs not supported
- Code portability issues
  - Compiler-specific extensions to C/C++
  - Buggy, complex, and fragile makefiles
  - Pre-processor definitions and conditional compilation
  - Undefined behaviours of C/C++ programs

4. Analyzing Results

- No tool is perfect
  - Will not find every kind of bug
  - Will not always find every instance of a kind of bug
    - ≥ 70%
- You will always find some bugs
  - If you don't, something is incorrect
- False positives are the main problem here
  - Diagnostic quality
  - Documentation quality

Murphy's Eighth Law: If everything seems to be going well, you have obviously overlooked something. – Arthur Bloch, Murphy's Law and Other Reasons Why Things Go Wrong, 1977.

Conclusion

- From formal methods to advanced heuristics
  - Seems to be the only way to scale right now
  - No one tries to do a complete analysis
  - But soundness is still very important
- Static and dynamic tools
  - should be used in conjunction
  - should be integrated into the development process
- Debugging other's people code
  - Not a good idea...

Conclusion

- Verifying C/C++ programs is a huge challenge
  - Undefined behaviours, pointers, compiler-specific extensions, etc.
  - No verification tool can reduce the risk enough for sensitive applications
  - It is based on a best-effort strategy
    - No statistics on assurance
  - Use C/C++ only if you really have to
    - Restrict language usage (MISRA, JSF++)
    - Use test cases and verification tools
- Modern, managed languages like Java or C# (keep it managed!) are much preferable

Frederic.Michaud@drdc-rddc.gc.ca
Frederic.Painchaud@drdc-rddc.gc.ca

DEFENCE R&D DÉFENSE