PA193 - Secure coding principles and practices

Static analysis of source code

Petr Švenda svenda@fi.muni.cz

CROCS
Centre for Research on Cryptography and Security

Overview

- Lecture: problems (demo), prevention
  - example of problems
  - types of static analysis
  - common types of errors, problem of false positives
  - design for testability
  - static analysis tools
- Labs
  - run and fix results from static checker
  - write own additional rule to checker

PA193 - exams

- 11.12. 18:00 A319
- 16.12. 18:00 B411
- 19.12. 10:00 A319

PROBLEM

Cost of insecure software

- Increased risk and failures due to generally increased usage of computers
- Fixing bug in released version is more expensive
  - testing, announcements,
- Liability laws
  - need to notify, settlement...
- Reputation loss
- Cost of defense is decreasing
  - better training (like this course ☺), automated tools, development methods

What is wrong with this code?

```
year = ORIGINYEAR; /* = 1980 */
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

Microsoft's Zune bug

- December 31st 2008
- Simultaneous fail of thousands music players
- http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/
- Highly embarrassing (blogs)
- Contributed to discontinuation

```
year = ORIGINYEAR; /* = 1980 */
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

What is wrong with this code?

```
network_receive(in_packet, &in_packet_len); // TLV packet
in = in_packet + 3;




out_packet = malloc(1 + 2 + length);
out = out_packet + 3;

memcpy(out, in, length);




network_transmit(out_packet);
```

OpenSSL Heartbeat – “packet repeater”

```
network_receive(in_packet, &in_packet_len); // TLV packet
in = in_packet + 3;
```

unsigned char* in


```
out_packet = malloc(1 + 2 + length);
out = out_packet + 3;

memcpy(out, in, length);
```

unsigned char* out


```
network_transmit(out_packet);
```

Problem?

```
network_receive(in_packet, &in_packet_len); // TLV packet
in = in_packet + 3;
```

unsigned char* in


```
out_packet = malloc(1 + 2 + length);
out = out_packet + 3;

memcpy(out, in, length);
```

in_packet_len != length + 3

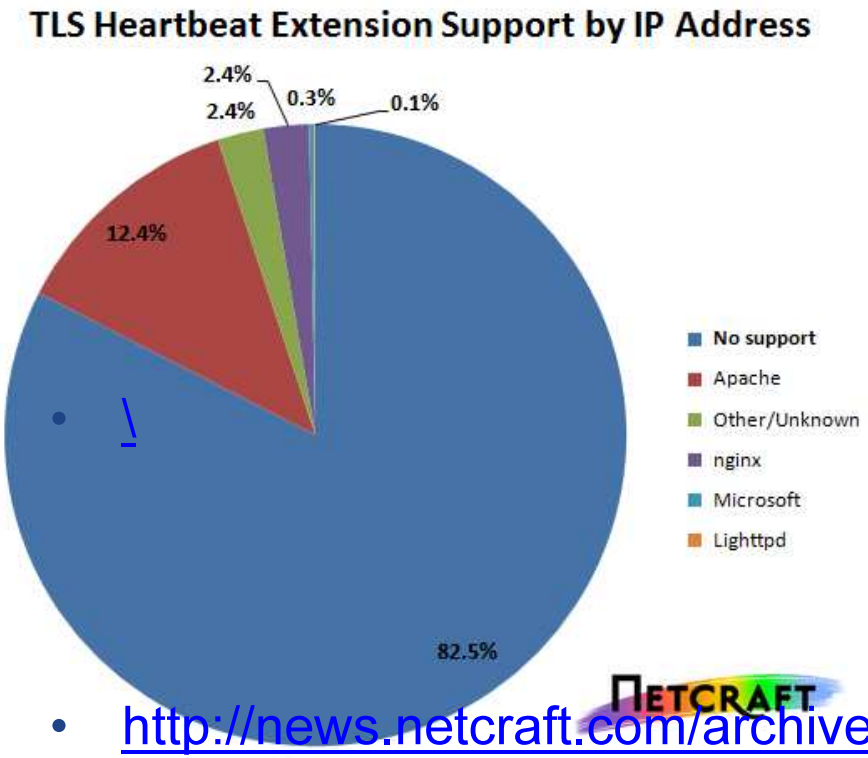
unsigned char* out


```
network_transmit(out_packet);
```

Problem!

How serious the bug was?

17% SSL web servers (OpenSSL 1.0.1)
Twitter, GitHub, Yahoo, Tumblr, Steam, DropBox, DuckDuckGo…
https://seznam.cz, https://fi.muni.cz …



- \
- http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html

Defensive programming

- Term coined by Kernighan and Plauger, 1981
  - "writing the program so it can cope with small disasters"
  - talked about in introductory programming courses
- Practice of coding with the mind-set that errors are inevitable and something will always go wrong
  - prepare program for unexpected behavior
  - prepare program for easier bug diagnostics
- Defensive programming targets mainly unintentional errors (not intentional attacks)
- Increasingly given security connotation

“Security features != Secure features”

- *“Security features != Secure features”*
  - *Howard and LeBlanc, 2002*
- *“Writing security features, although important, is only 10% of the workload of creating secure code. The other 90% of the coding work is meant to ensure that all non-security codebase is secure.”*
  - *Sullivan, Balinsky, 2012*
- *“Reliable software does what it is supposed to do. Secure software does what it is supposed to do, and nothing else.”*
  - *Ivan Arce*

STATIC AND DYNAMIC ANALYSIS

How to find bugs in code?

- Manual dynamic testing
  - running program, observe expected output
- Manual analysis of code
  - code review
- Automated analysis of code without compilation
  - static analysis (pattern matching, symbolic execution)
- Automated analysis of code with execution
  - dynamic analysis (running code)
- Automated testing of inputs (fuzzing)

Approaches for automated code review

- Formal methods (mathematical verification)
  - requires mathematical model and assertions
  - often requires modeling the system as finite state machine
    - verification of every state and transition
- Code metrics
  - help to identify potential hotspots (complex code)
  - e.g., Cyclomatic complexity (number of linearly indep. paths)
- Review and inspection
  - tries to find suspicious patterns
  - automated version of human code review

Microsoft's Secure Development Lifecycle

Training	Requirements	Design	Implementation	Verification	Release	Response
1. Core Security Training	2. Establish Security Requirements	5. Establish Design Requirements	8. **Use Approved Tools**	11. Perform Dynamic Analysis	14. Create an Incident Response Plan	
	3. Create Quality Gates/Bug Bars	6. Perform Attack Surface Analysis/ Reduction	9. **Deprecate Unsafe Functions**	12. Perform Fuzz Testing	15. Conduct Final Security Review	Execute Incident Response Plan
	4. Perform Security and Privacy Risk Assessments	7. Use Threat Modeling	10. **Perform Static Analysis**	13. Conduct Attack Surface Review	16. Certify Release and Archive	

Taken from http://www.microsoft.com/security/sdl/process/implementation.aspx

Cigital Touchpoints methodology

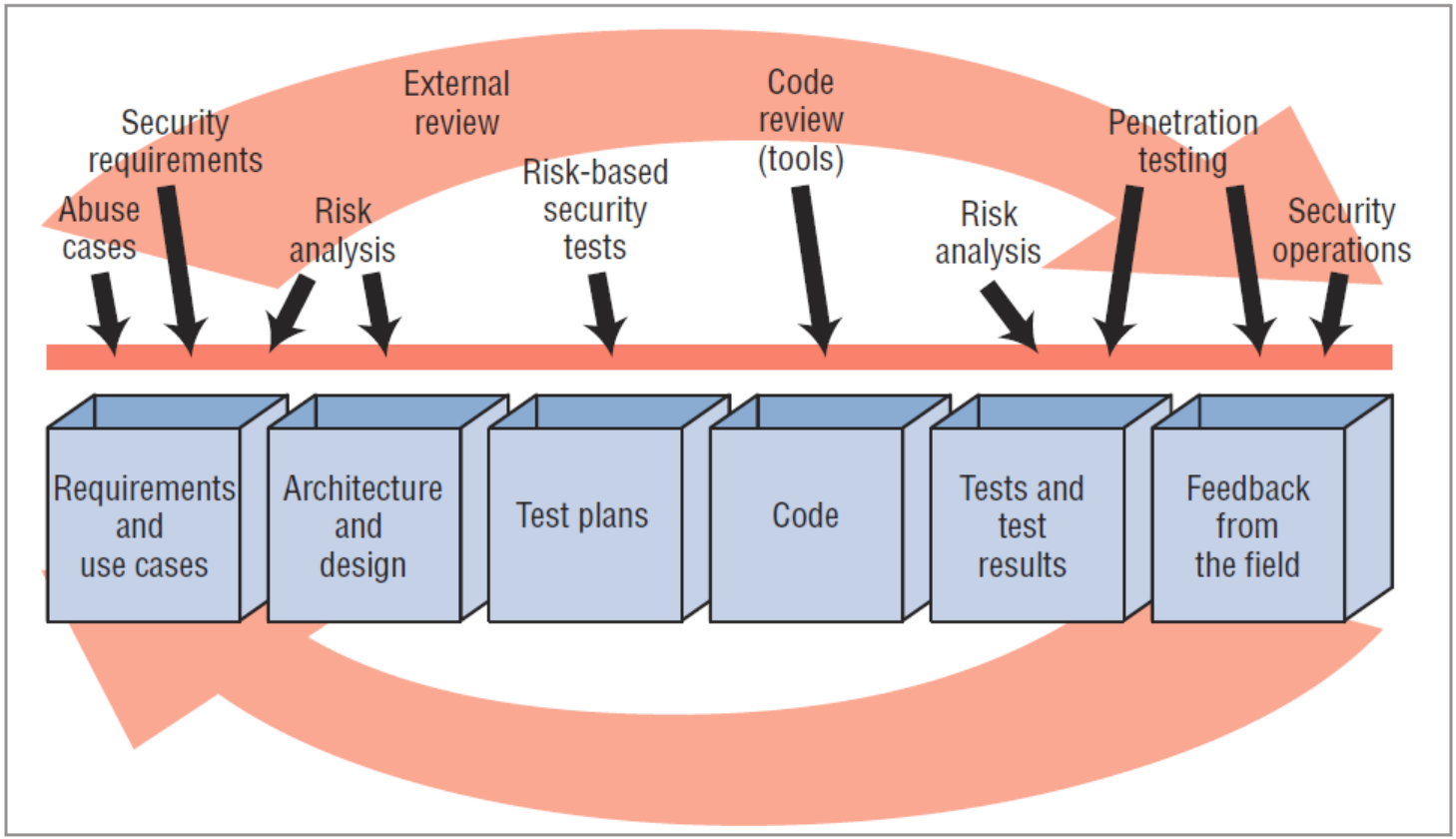
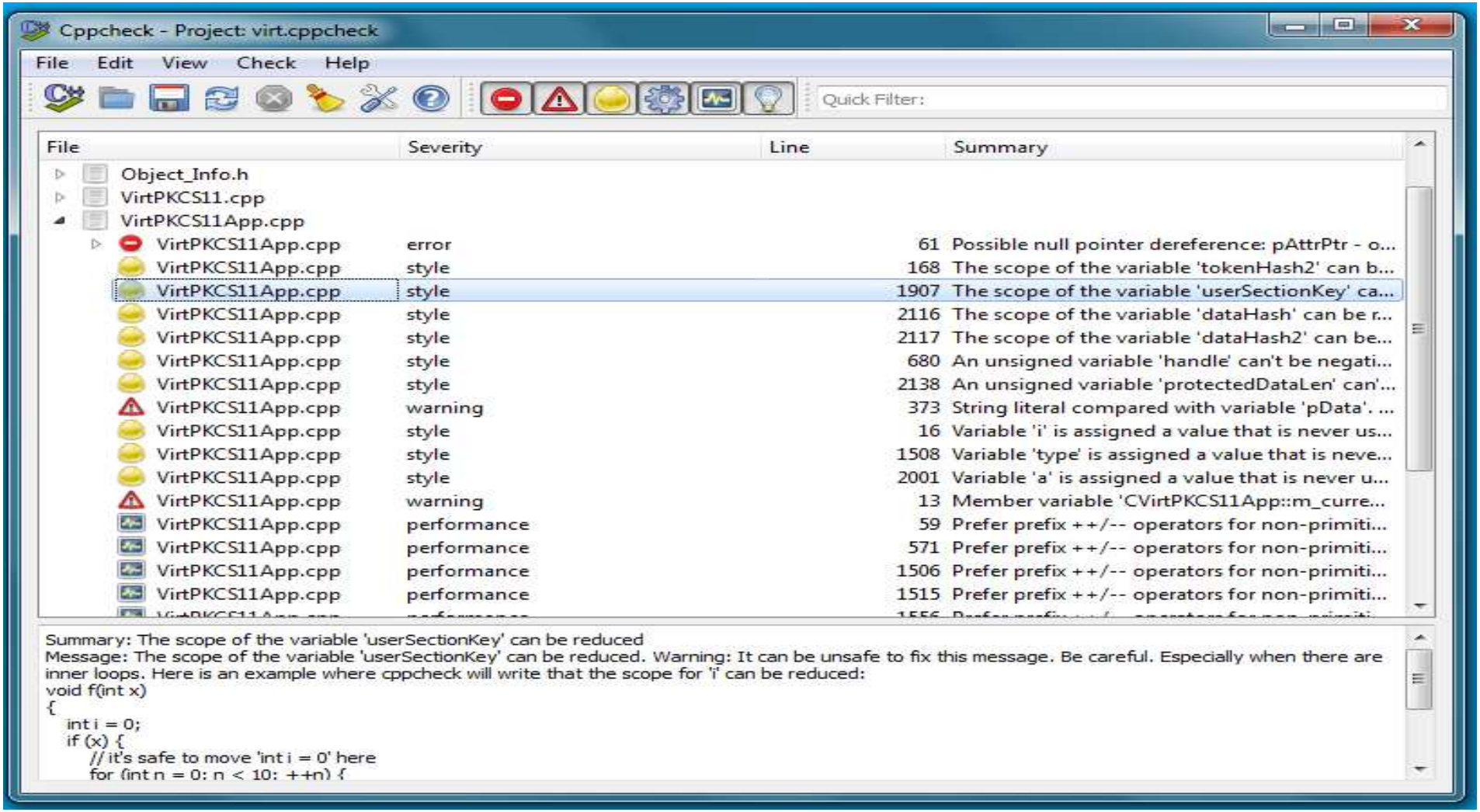


Figure 1. The Cigital Touchpoints methodology. Software security best practices (arrows) applied to various software artifacts (boxes).

Static vs. dynamic analysis

- Static analysis
  - examine program's code without executing it
  - can examine both source code and compiled code
    - source code is easier to understand (more metadata)
  - can be applied on unfinished code
  - manual code audit is kind of static analysis
- Dynamic analysis
  - code is executed (compiled or interpreted)
  - input values are supplied, internal memory is examined

Example of output produced by analyzer



Types of static analysis

- Type checking – performed by compiler
- Style checking – performed by automated tools
- Program formal verification
  - annotations & verification of specified properties
- Bug finding / hunting
  - between style checking and verification
  - more advanced static analysis
  - aim to infer real problem, not only pattern match
- Security Review
  - previous possibilities with additional support for review

Static analysis - techniques

- Structural rules (unwanted functions / patterns)
  - deprecated functions (e.g., gets)
  - fixed size arrays (e.g., `char buff[100]`)
- Trace of interesting data throw program
  - propagation of tainted data (user input → exec(data))
  - match of possible lengths for input / output data

Type checking

- Type checking – performed by compiler
  - errors against language rules prevents compilation
  - warnings usually issued when problematic type manipulation occur
  - false positives (short = int = short;)
- Security problems due to wrong types
  - string format vulnerabilities
  - type overflow → buffer overflow
  - data loss (bigger type to smaller type)
- More on type checking later with compiler warnings

Style checking

- Style checking – performed by automated tools
  - set of required code rules
- Separate tools
  - MS style checker
  - Unix: lint tool (http://www.unix.com/man-page/FreeBSD/1/lint)
  - Checkstyle
  - PMD (http://pmd.sourceforge.net/)
  - Google C++ style checker: C++lint
    - http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
    - http://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py
- Compiler warnings `gcc -Wall gcc -Wextra`

Program verification

- Prove particular program property
  - e.g., all dynamically allocated memory is always freed
- Requires mathematical model and assertions
- Often requires modeling the system as finite state machine
  - verification of every state and transition
- (Outside the scope of this presentation)

Bug finding

- No language errors != secure program
  - finding bugs, even when language permits it
- Examples:
  - Buffer overflow possible?
  - User input formated into system() call?
  - Hard-code secrets?
- Must keep false positives low
  - do not report as a bug something which isn't
  - there is simply too many potential problems
- Tools: FindBugs, PREfast, Coverity...

Security analysis and review

- Usage of analysis tool to perform security review
  - usually multiple tools are used during the process
- (Will be covered in later lecture(s))

Static analysis limitations

- Overall program architecture is not understand
  - sensitivity of program path
  - impact of errors on other parts
- Application semantics is not understand
  - Is string returned to the user? Can string also contain passwords?
- Social context is not understand
  - Who is using the system? High entropy keys encrypted under short guessable password?

Problem of false positives/negatives

- False positives
  - errors reported by a tool that are not errors in fact
  - too conservative analysis
  - inaccurate model used for analysis
  - annoying, more code needs to be checked, less readable output, developers tend to have as an excuse
- False negatives
  - real errors NOT reported by a tool
  - missed problems, missing rules for detection

False positives – limits of static analysis

```
void foo()
{
  char a[10];
  a[20] = 0;
}
```

```
d:\StaticAnalysis>cppcheck example.cpp
Checking example.cpp...
[example.cpp:4]: (error) Array 'a[10]' accessed at index 20, which
                 is out of bounds.
```

- When foo() is called, always writes outside buffer

False positives – limits of static analysis

```
int x = 0;
int y = 3;
void foo()
{
  char a[10];
  if (x + y == 2) {
    a[20] = 0;
  }
}
```

problematic assignment put inside condition

```
d:\StaticAnalysis>cppcheck example.cpp
Checking example.cpp...
[example.cpp:7]: (error) Array 'a[10]' accessed at index 20, which
                         is out of bounds.
```

- For `x + y != 2` false positive
- But analyzer cannot be sure about x & y values

False positives – limits of static analysis

```
const int x = 0;
const int y = 3;
void foo()
{
  char a[10];
  if (x + y == 2) {
    a[20] = 0;
  }
}
```

const added (same for #define)

```
d:\StaticAnalysis>cppcheck example.cpp
Checking example.cpp...
```

```
d:\StaticAnalysis>cppcheck --debug example.cpp
Checking example.cpp...

##file example.cpp
1:
2:
3:
4: void foo ( )
5: {
6: char a@3 [ 10 ] ;
7:
8:
9:
10: }
```

- No problem detected – constants are evaluated and condition completely removed

False positives – limits of static analysis

```
void foo2(int x, int y) {
  char a[10];
  if (x + y == 2) {
    a[20] = 0;
  }
}
int main() {
  foo2(0, 3);
  return 0;
}
```

```
d:\StaticAnalysis>cppcheck --debug example.cpp
Checking example.cpp...

##file example.cpp
1: void foo2 ( int x@1 , int y@2 ) {
2: char a@3 [ 10 ] ;
3: if ( x@1 + y@2 == 2 ) {
4: a@3 [ 20 ] = 0 ;
5: }
6: }
7: int main ( ) {
8: foo2 ( 0 , 3 ) ;
9: return 0 ;
10:}

[example.cpp:4]: (error) Array 'a[10]' accessed at index 20,
                         which is out of bounds.
```

- Whole program is not compiled and evaluated

Always design for testability

- *"Code that isn't tested doesn't work - this seems to be the safe assumption."* Kent Beck
- Code written in a way that is easier to test
  - proper decomposition, unit tests, mock objects
  - source code annotations (with subsequent analysis)
- References
  - https://en.wikipedia.org/wiki/Design_For_Test
  - http://www.agiledata.org/essays/tdd.html

BUILD-IN COMPILER ANALYSIS

MSVC flags

```
#include <iostream>
using namespace std;
int main(void) {
  int low_limit = 0;
  for (unsigned int i = 10; i >= low_limit; i--) {
    cout << ".";
  }
  return 0;
}
```

- warning C4018: '>=' : **signed/unsigned** mismatch

Warnings – how compiler signals troubles

- MSVC /W n
  - /W 0 disables all warnings
  - /W 1 & /W 2 basic warning
  - /W 3 recommended production purposes (default)
  - /W 4 recommended for all compilations, ensure the fewest possible hard-to-find code defects
  - /Wall == /W4 + extra
- GCC -Wall, -Wextra
- Treat warnings as errors
  - GCC –Werror, MSVC /WX
  - forces you to fix all warnings, but slightly obscure nature of problem

Recommendations for MSVC CL

- Compile with higher warnings /W4
- Control and fix especially integer-related warnings
  - warning C4018: '>=' : signed/unsigned mismatch
    - comparing signed and unsigned values, signed value must be converted to unsigned
  - C4244, C4389 – possible loss of data because of truncation or signed&unsigned variables operation
- If existing code is inspected, look for
  - `#pragma warning (disable, Cxxxx)` where xxxx is above
- Use compiler /RTC flag

warning C4018: '>=' : signed/unsigned mismatch

- What will be the output of following code?
  - string "x > y"
  - but also compiler warning C4018

```
#include <iostream>
using namespace std;
int main(void) {
  int x = -100;
  unsigned int y = 100;
  if (x > y) { cout << "x > y"; }
  else { cout << "y >= x"; }

  return 0;
}
```

Recommendations for GCC

- GCC –Wconversion
  - warn about potentially problematic conversions
  - fixed → floating point, signed → unsigned, ...
- GCC –Wsign-compare
  - signed → unsigned producing incorrect result
  - **`warning: comparison between signed and unsigned integer expressions [-Wsign-compare]`**
  - see http://stackoverflow.com/questions/16834588/wsign-compare-warning-in-g for example of real problem
- Runtime integer error checks using **`-ftrapv`**
  - trap function called when signed overflow in addition, subs, mult. occur
  - but significant performance penalty (continuous overflow checking) ☹

Try this at home!

GCC -ftrapv

```c
/* compile with gcc -ftrapv <filename> */
#include <signal.h>
#include <stdio.h>
#include <limits.h>

void signalHandler(int sig) {
  printf("Overflow detected\n");
}

int main() {
  signal(SIGABRT, &signalHandler);

  int largeInt = INT_MAX;
  int normalInt = 42;
  int overflowInt = largeInt + normalInt;  /* should cause overflow */

  /* if compiling with -ftrapv, we shouldn't get here */
  return 0;
}
```

http://stackoverflow.com/questions/5005379/c-avoiding-overflows-when-working-with-big-numbers

STATIC ANALYSIS TOOLS

Static analysis tools

- List of static checkers
  - http://spinroot.com/static/
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
  - https://security.web.cern.ch/security/recommendations/en/code_tools.shtml
- We will be interested in C/C++ checkers
  - but tools exists for almost any language

Both free and commercial tools

- Commercial tools
  - PC-Lint (Gimpel Software)
  - Klocwork Insight (Klocwork)
  - Coverity Prevent (now under HP)
  - Microsoft PREfast (included in Visual Studio)
- Free tools
  - Rough Auditing Tool for Security (RATS) http://code.google.com/p/rough-auditing-tool-for-security/
  - CppCheck http://cppcheck.sourceforge.net/
  - Flawfinder http://www.dwheeler.com/flawfinder/
  - Splint http://www.splint.org/
  - FindBugs http://findbugs.sourceforge.net (for Java programs)
  - Doxygen's call graphs from source http://www.stack.nl/~dimitri/doxygen/
  - ...

Flawfinder

- Last version 1.27 (2007-01-16)
- Download at http://www.dwheeler.com/flawfinder/
- Build by `setup.py build`
- Install by `setup.py install`
- `/build/scripts***/flawfinder.py`
- `flawfinder.py --context --html source_dir`

Flawfinder - example

```
  strncat(d,s,10);
source\test.c:58:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated (it could cause a
  crash if unprotected).
  n = strlen(d);
source\test.c:64:  [1] (buffer) MultiByteToWideChar:
  Requires maximum length in CHARACTERS, not bytes. Risk is very low,
  the length appears to be in characters not bytes.
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof(wszUserName)/sizeof(
wszUserName[0]));
source\test.c:66:  [1] (buffer) MultiByteToWideChar:
  Requires maximum length in CHARACTERS, not bytes. Risk is very low,
  the length appears to be in characters not bytes.
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof wszUserName /sizeof(
wszUserName[0]));

Hits = 36
Lines analyzed = 117 in 0.93 seconds (273 lines/second)
Physical Source Lines of Code (SLOC) = 80
Hits@level = [0]   0 [1]   9 [2]   7 [3]   3 [4]  10 [5]   7
Hits@level+ = [0+]  36 [1+]  36 [2+]  27 [3+]  20 [4+]  17 [5+]   7
Hits/KSLOC@level+ = [0+] 450 [1+] 450 [2+] 337.5 [3+] 250 [4+] 212.5 [5+] 87.5
Suppressed hits = 2 (use --neverignore to show them)
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

C:\Program Files\Flawfinder\build\scripts-2.5>flawfinder.py --context source
```

Splint

- Secure Programming Lint
- Annotation-Assisted Lightweight Static Checking
- http://www.splint.org/
  - standard static analyzer
  - possibility to add annotations
- Last version 3.1.2 (2007)
- Splint overview
  - http://www.slideshare.net/UlissesCosta/splint-the-c-code-static-checker

RATS

- Rough Auditing Tool for Security (RATS)
  - http://code.google.com/p/rough-auditing-tool-for-security/
- Windows and Linux support
- Previous version 2.3 (2009)
- Last version 2.4 (December 2013)

Cppcheck

- A tool for static C/C++ code analysis
  - Open-source freeware, http://cppcheck.sourceforge.net/
  - Online demo http://cppcheck.sourceforge.net/demo/
- Last version 1.66 (2014-08-02)
- Used to find bugs in open-source projects (Linux kernel...)
- Command line & GUI version
- Standalone version, plugin into IDEs, version control...
  - Code::Blocks, Codelite, Eclipse, Jenkins...
  - Tortoise SVN
  - not Visual Studio ☹
- Cross platform (Windows, Linux)
  - `sudo apt-get install cppcheck`

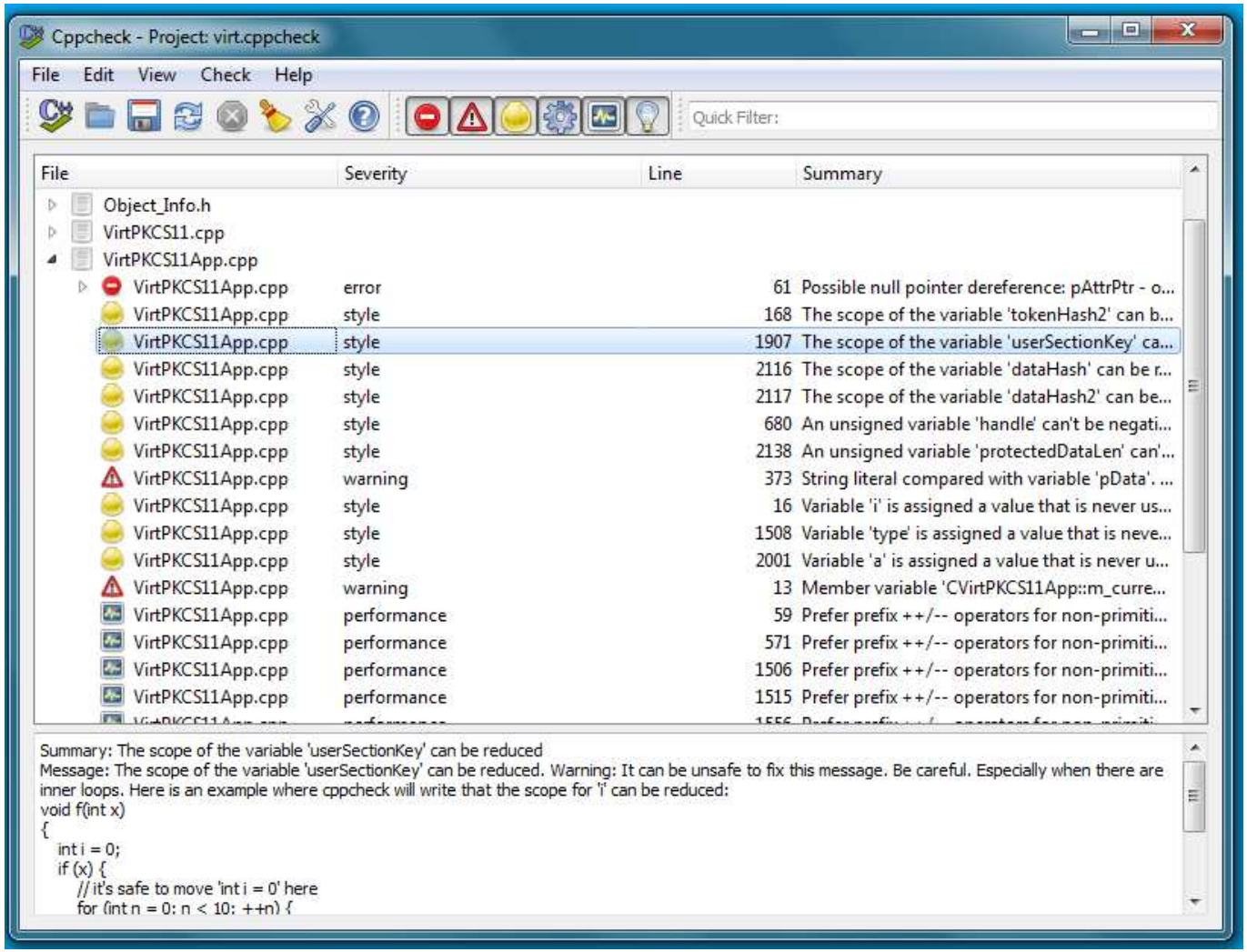
Cppcheck – what is checked?

- Bound checking for array overruns
- Suspicious patterns for class
- Exceptions safety
- Memory leaks
- Obsolete functions
- sizeof() related problems
- String format problems...
- See full list http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page#Checks

Cppcheck – categories of problems

- error – when bugs are found
- warning - suggestions about defensive programming to prevent bugs
- style - stylistic issues related to code cleanup (unused functions, redundant code, constness...)
- performance - suggestions for making the code faster.
- portability - portability warnings. 64-bit portability. code might work different on different compilers. etc.
- information - Informational messages about checking problems

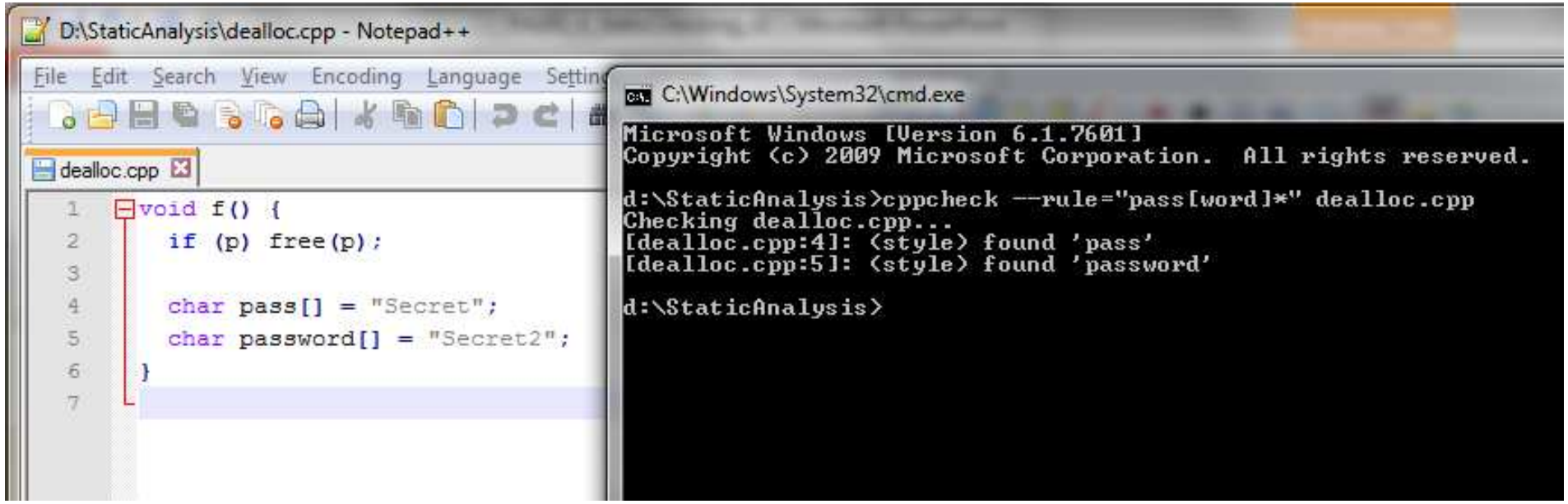
Cppcheck


Cppcheck - Project: virt.cppcheck
File Edit View Check Help
Quick Filter:
File Severity Line Summary
Object_Info.h
VirtPKCS11.cpp
VirtPKCS11App.cpp
VirtPKCS11App.cpp error 61 Possible null pointer dereference: pAttrPtr - o...
VirtPKCS11App.cpp style 168 The scope of the variable 'tokenHash2' can b...
VirtPKCS11App.cpp style 1907 The scope of the variable 'userSectionKey' ca...
VirtPKCS11App.cpp style 2116 The scope of the variable 'dataHash' can be r...
VirtPKCS11App.cpp style 2117 The scope of the variable 'dataHash2' can be...
VirtPKCS11App.cpp style 680 An unsigned variable 'handle' can't be negati...
VirtPKCS11App.cpp style 2138 An unsigned variable 'protectedDataLen' can'...
VirtPKCS11App.cpp warning 373 String literal compared with variable 'pData'. ...
VirtPKCS11App.cpp style 16 Variable 'i' is assigned a value that is never us...
VirtPKCS11App.cpp style 1508 Variable 'type' is assigned a value that is neve...
VirtPKCS11App.cpp style 2001 Variable 'a' is assigned a value that is never u...
VirtPKCS11App.cpp warning 13 Member variable 'CVirtPKCS11App::m_curre...
VirtPKCS11App.cpp performance 59 Prefer prefix ++/-- operators for non-primiti...
VirtPKCS11App.cpp performance 571 Prefer prefix ++/-- operators for non-primiti...
VirtPKCS11App.cpp performance 1506 Prefer prefix ++/-- operators for non-primiti...
VirtPKCS11App.cpp performance 1515 Prefer prefix ++/-- operators for non-primiti...
Summary: The scope of the variable 'userSectionKey' can be reduced
Message: The scope of the variable 'userSectionKey' can be reduced. Warning: It can be unsafe to fix this message. Be careful. Especially when there are inner loops. Here is an example where cppcheck will write that the scope for 'i' can be reduced:
void f(int x)
{
int i = 0;
if (x) {
// it's safe to move 'int i = 0' here
for (int n = 0; n < 10; ++n) {

Cppcheck – simple custom rules

- User can write own regular expression-based rules
  - Perl Compatible Regular Expressions www.pcre.org
  - limited only to simpler analysis
  - executed over *simplified* code (code after preprocessing)
    - http://sourceforge.net/projects/cppcheck/files/Articles/writing-rules-2.pdf
- Regular expression can be supplied on command line
  - `cppcheck.exe --rule=".+" file.cpp`
    - match any code, use to obtain simplified code
  - `cppcheck.exe --rule="pass[word]*" file.cpp`
    - match any occurrence of pass or password or passwordword...
- Or via XML file (for stable repeatedly used rules)

`cppcheck.exe` --rule="pass[word]*" file.cpp



- `cppcheck.exe` --rule="if \(p \) { free \(p \) ; }" file.cpp
  - will match only pointer with name 'p'

Cppcheck – simple custom rules (XML)

- XML file with regular expression and information
  - pattern to search for
  - information displayed on match

```
<?xml version="1.0"?>
<rule>
 <tokenlist>LIST</tokenlist>
 <pattern>PATTERN</pattern>
 <message>
  <id>ID</id>
  <severity>SEVERITY</severity>
  <summary>SUMMARY</summary>
 </message>
</rule>
```

```
<?xml version="1.0"?>
<rule version="1">
<pattern>if \( p \) { free \( p \) ; }</pattern>
<message>
<id>redundantCondition</id>
<severity>style</severity>
<summary>Redundant condition. It is valid
  to free a NULL pointer.
</summary>
</message>
</rule>
```

Example taken from http://sourceforge.net/projects/cppcheck/files/Articles/writing-rules-1.pdf/download

Cppcheck – complex custom rules

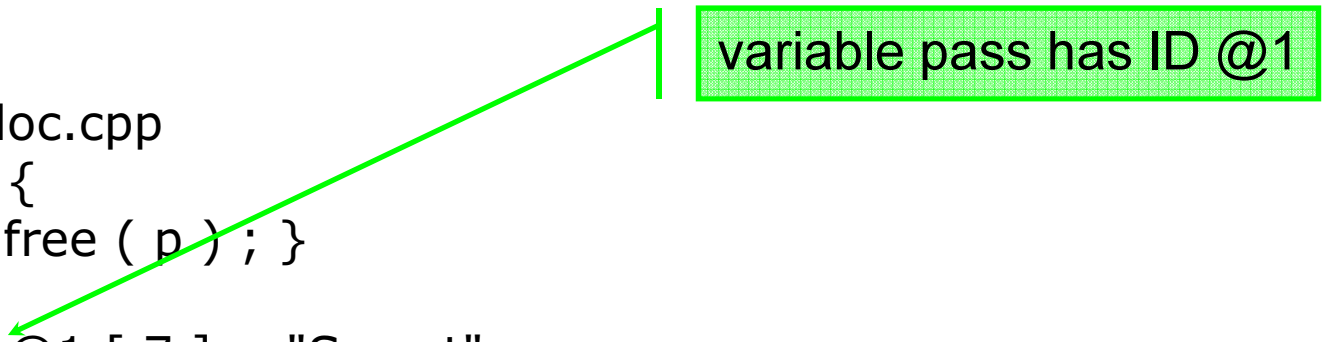
- Based on execution of user-supplied C++ code
  - possible more complex analysis

1. Use `cppcheck.exe` --debug file.cpp
   - outputs simplified code including Cppcheck's internal variable unique ID
2. Write C++ code fragment performing analysis
3. Recompile Cppcheck with new rule and execute

- Read more details (writing-rules-2 & writing-rules-3)
  - http://sourceforge.net/projects/cppcheck/files/Articles/

Custom rules – obtaining variable ID

```
d:\StaticAnalysis>cppcheck.exe --debug dealloc.cpp
Checking dealloc.cpp...
```



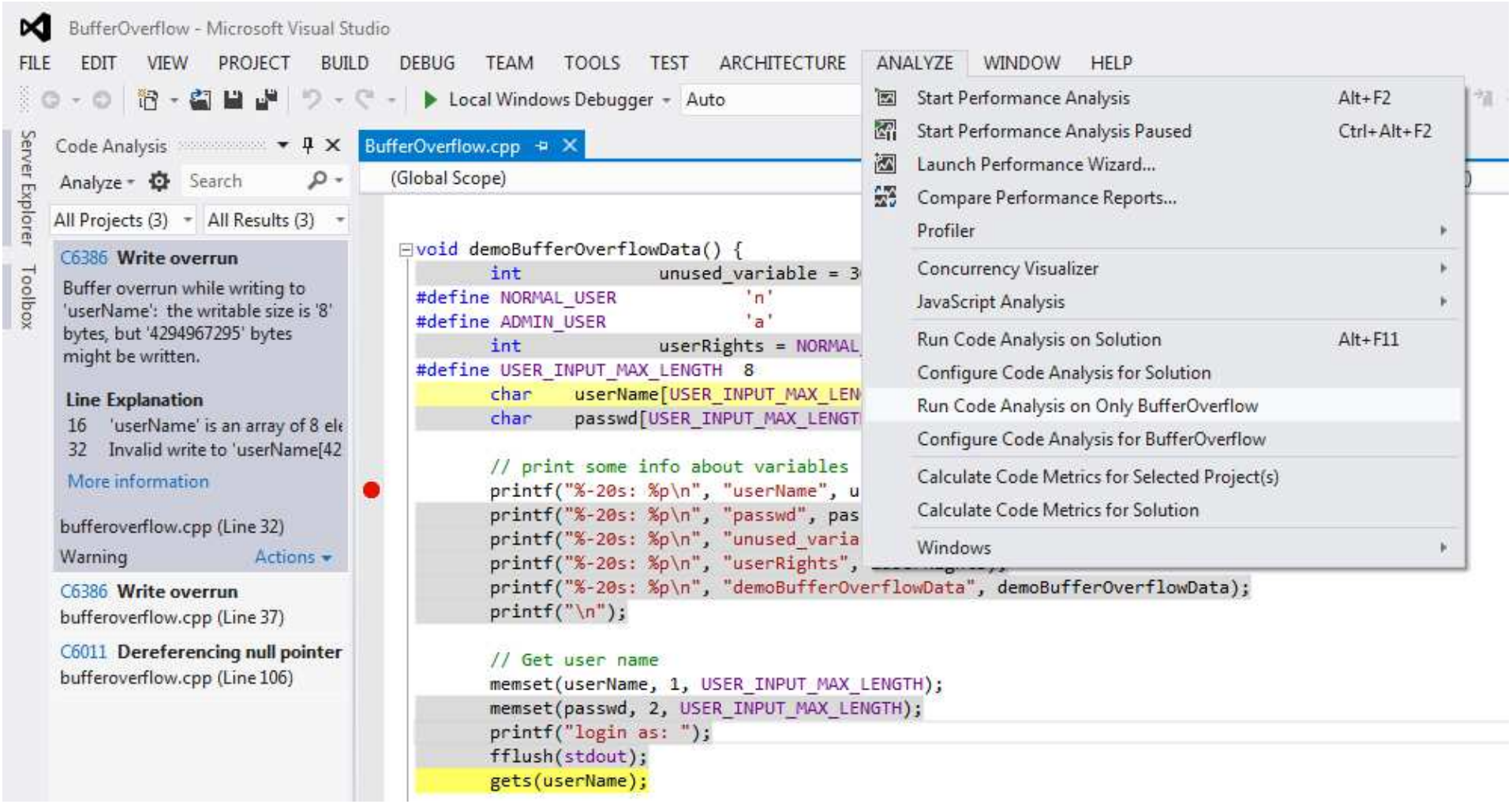

```
##file dealloc.cpp
1: void f ( ) {
2: if ( p ) { free ( p ) ; }
3:
4: char pass@1 [ 7 ] = "Secret" ;
5: char password@2 [ 8 ] = "Secret2" ;
6: }
```

```
void CheckOther::dealloc() {
  // Loop through all tokens
  for (const Token *tok = _tokenizer->tokens(); tok; tok = tok->next()) {
    // Is there a condition and a deallocation?
    if (Token::Match(tok, "if ( %var% ) { free ( %var% ) ; }")) {
      // Get variable name used in condition:
      const std::string varname1 = tok->strAt(2);
      // Get variable name used in deallocation:
      const std::string varname2 = tok->strAt(7);
      // Is the same variable used?
      if (varname1 == varname2) {
        // report warning
        deallocWarning(tok);
      }
    }
  }
}
// Report warning
void CheckOther::deallocWarning() {
  reportError(tok, // location
  Severity::warning, // severity
  "dealloc", // id
  "Redundant condition"); // message
}
```

pattern to match %var% will match any variable

reporting error in standard format

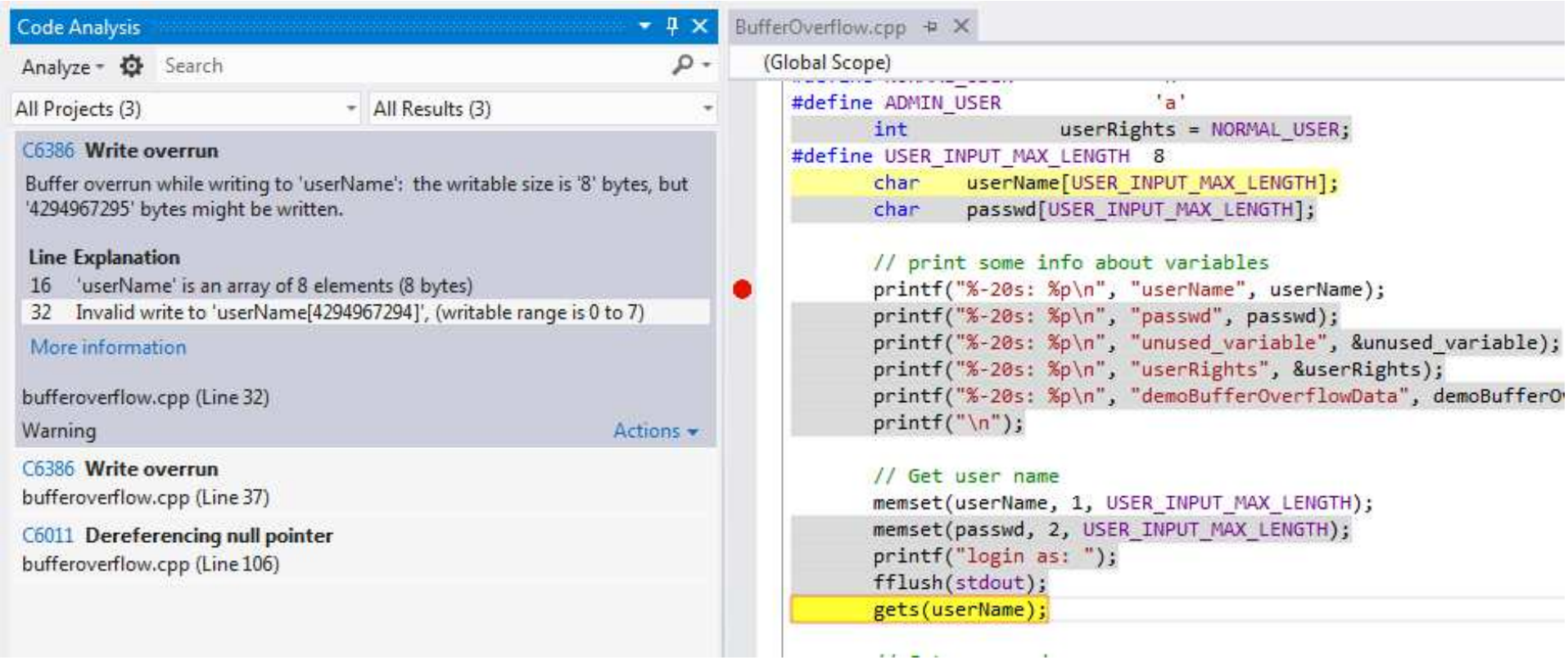
PREfast - Microsoft static analysis tool



PREfast - Microsoft static analysis tool

- Visual Studio Ultimate and Premium Editions
- Documentation for PREfast
  - http://msdn.microsoft.com/en-us/library/windows/hardware/gg487351.aspx
- PREfast tutorial
  - http://www.codeproject.com/Articles/167588/Using-PREfast-for-Static-Code-Analysis
- Can be enabled on every build
  - not enabled by default, time consuming
- Can be extended by source code annotation (SAL)
  - (next lecture)

PREfast – example bufferOverflow

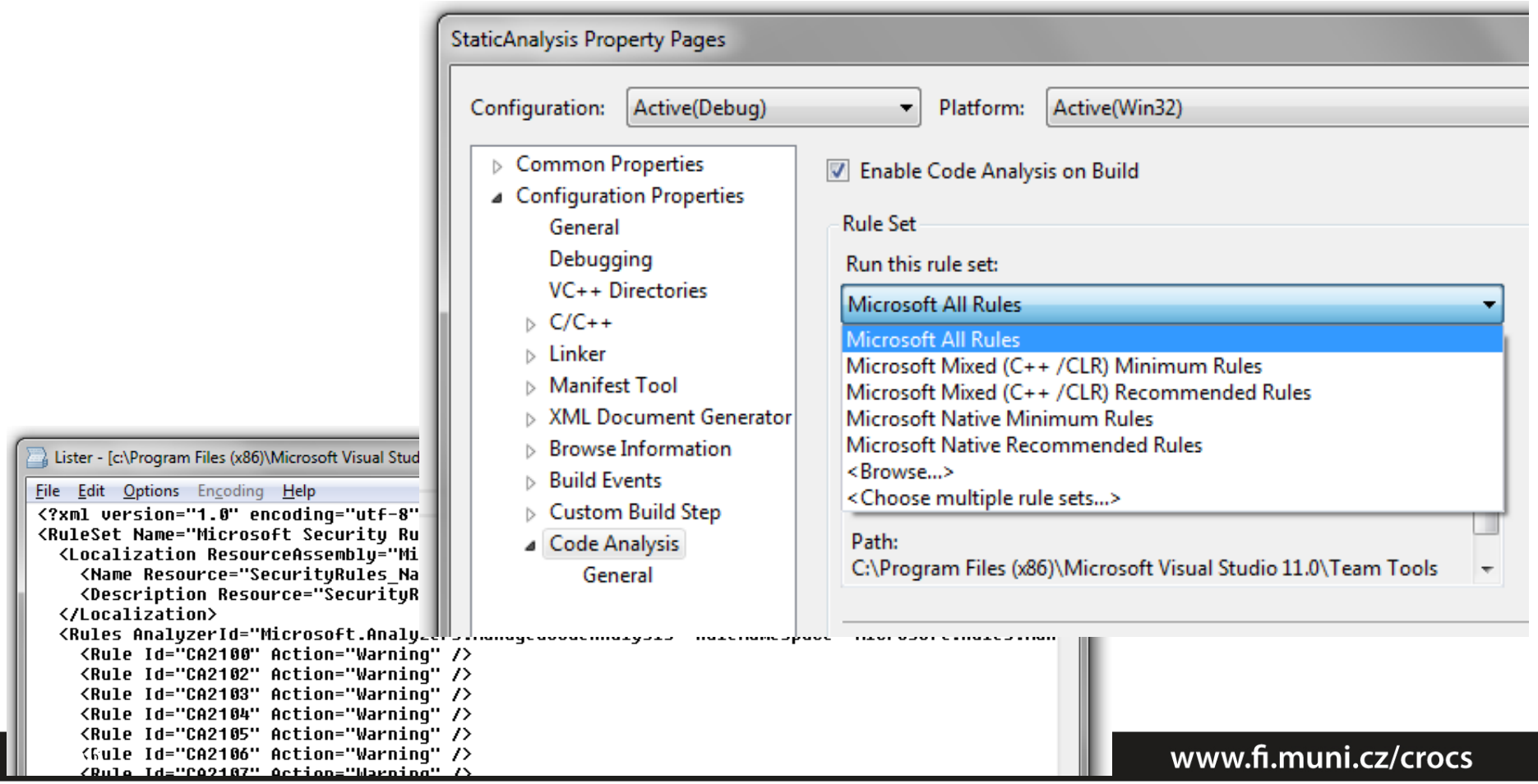


PREfast – what can be detected

- Potential buffer overflows
- Memory leaks, uninitialized variables
- Excessive stack usage
- Resources – release of locks...
- Incorrect usage of selected functions
- List of all code analysis warnings
  http://msdn.microsoft.com/en-us/library/a5b9aa09.aspx

PREfast settings (VS 2012)

- http://msdn.microsoft.com/en-us/library/ms182025.aspx

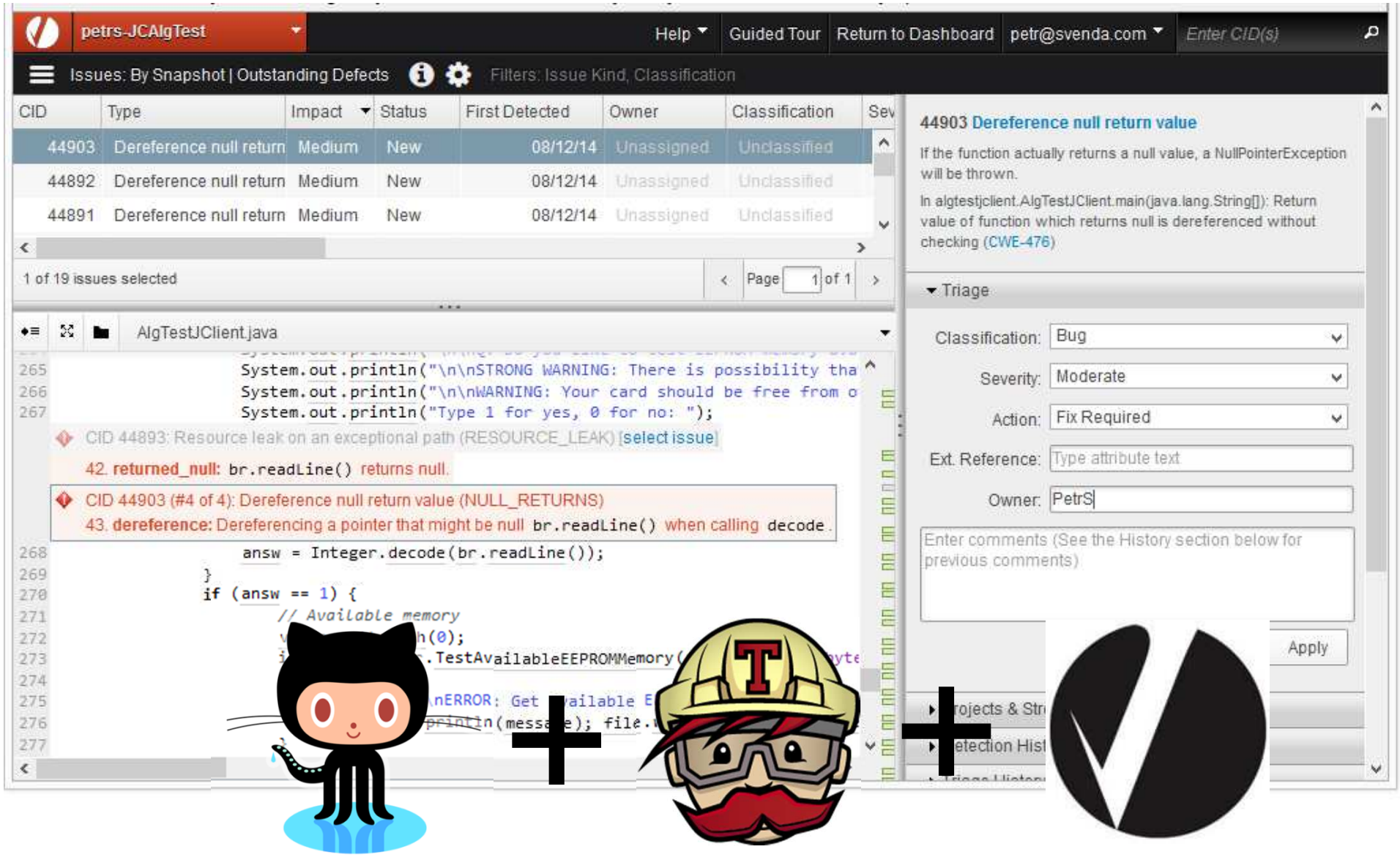


PREfast & MSVC /analyze

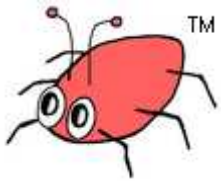
- Enables code analysis and control options
  - http://msdn.microsoft.com/en-us/library/ms173498.aspx
- Some analysis rules work only for managed code (C#, VB...)
- Available rule sets
  - http://msdn.microsoft.com/en-us/library/dd264925%28v=vs.120%29.aspx
- Writing custom rules
  - http://msdn.microsoft.com/en-us/library/dd380660%28v=vs.120%29.aspx

Coverity (free for open-source)

- Commercial static & dynamic analyzer
- Free for C/C++ & Java open-source projects
- https://scan.coverity.com/
- Process
  - Register at scan.coverity.com (GitHub account possible)
  - Download Coverity build tool for your platform
    - Quality and Security Advisor
  - Build your project with cov-build
    - cov-build --dir cov-int <build command>
  - Zip and submit build for analysis
- Can be integrated with Travis CI (continuous integration)
  - https://scan.coverity.com/travis_ci


petrs-JCAlgTest
Help
Guided Tour
Return to Dashboard
petr@svenda.com
Enter CID(s)
Issues: By Snapshot | Outstanding Defects
Filters: Issue Kind, Classification
CID
Type
Impact
Status
First Detected
Owner
Classification
44903 Dereference null return Medium New 08/12/14 Unassigned Unclassified
44892 Dereference null return Medium New 08/12/14 Unassigned Unclassified
44891 Dereference null return Medium New 08/12/14 Unassigned Unclassified
1 of 19 issues selected
Page 1 of 1
44903 Dereference null return value
If the function actually returns a null value, a NullPointerException will be thrown.
In algtestjclient.AlgTestJClient.main(java.lang.String[]): Return value of function which returns null is dereferenced without checking (CWE-476)
Triage
Classification: Bug
Severity: Moderate
Action: Fix Required
Ext. Reference: Type attribute text
Owner: PetrS
Enter comments (See the History section below for previous comments)
Apply
AlgTestJClient.java
System.out.println("\n\nSTRONG WARNING: There is possibility tha
System.out.println("\n\nWARNING: Your card should be free from o
System.out.println("Type 1 for yes, 0 for no: ");
CID 44893: Resource leak on an exceptional path (RESOURCE_LEAK) [select issue]
42. returned_null: br.readLine() returns null.
CID 44903 (#4 of 4): Dereference null return value (NULL_RETURNS)
43. dereference: Dereferencing a pointer that might be null br.readLine() when calling decode.
answ = Integer.decode(br.readLine());
}
if (answ == 1) {
// Available memory

FindBugs



- http://findbugs.sourceforge.net/
- Static analysis of Java programs
- Current version 2.0.2 (2012-12-02)
- Command-line, GUI, plugins into variety of tools
- Support for custom rules
- FindSecurityBugs
  - Additional detection rules for FindBugs
  - https://h3xstream.github.io/find-sec-bugs/bugs.htm

STATIC ANALYSIS IS NOT PANACEA

Cppcheck --enable=all

```
d:\StaticAnalysis>cppcheck --enable=all bufferOverflow.cpp
Checking bufferOverflow.cpp...
[bufferOverflow.cpp:26]: (style) Obsolete function 'gets' called. It is recommended to use
                         the function 'fgets' instead.
[bufferOverflow.cpp:31]: (style) Obsolete function 'gets' called. It is recommended to use
                         the function 'fgets' instead.
```

MSVC /W4

```
1>  BufferOverflow.cpp
1>bufferoverflow.cpp(32): warning C4996: 'gets': This function or variable may be unsafe.
          Consider using gets_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
1>  c:\program files (x86)\microsoft visual studio 11.0\vc\include\stdio.h(261) : see declaration of 'gets'
1>bufferoverflow.cpp(37): warning C4996: 'gets': This function or variable may be unsafe.
          Consider using gets_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
1>        c:\program files (x86)\microsoft visual studio 11.0\vc\include\stdio.h(261) : see declaration of 'gets'
1>bufferoverflow.cpp(78): warning C4996: 'strncpy': This function or variable may be unsafe.
          Consider using strncpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
1>  c:\program files (x86)\microsoft visual studio 11.0\vc\include\string.h(191) : see declaration of 'strncpy'
1>bufferoverflow.cpp(81): warning C4996: 'sprintf': This function or variable may be unsafe.
Consider using sprintf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
1>  c:\program files (x86)\microsoft visual studio 11.0\vc\include\stdio.h(357) : see declaration of 'sprintf'
```

```
fflush(stdout);
```

MSVC /analyze (PREfast)

```
1>  BufferOverflow.cpp
bufferoverflow.cpp(32): warning : C6386: Buffer overrun while writing to 'userName':
          the writable size is '8' bytes, but '4294967295' bytes might be written.
bufferoverflow.cpp(37): warning : C6386: Buffer overrun while writing to 'passwd':
          the writable size is '8' bytes, but '4294967295' bytes might be written.
```

From 2. lecture

Type overflow – example with dynalloc

```
typedef struct _some_structure {
        float   someData[1000];
} some_structure;

void demoDataTypeOverflow(int totalItemsCount, some_structure* pItem,
                          int itemPosition) {
 // See http://blogs.msdn.com/oldnewthing/archive/2004/01/29/64389.aspx
 some_structure* data_copy = NULL;
 int bytesToAllocation = totalItemsCount * sizeof(some_structure);
 printf("Bytes to alloca
 data_copy = (some_struc
 if (itemPosition >= 0 &
    memcpy(&(data_copy[i
 }
 else {
    printf("Out of bound
    return;
 }
 free(data_copy);
}
```

```
Cppcheck --enable=all
d:\StaticAnalysis>cppcheck --enable=all typeOverflow.cpp
Checking typeOverflow.cpp...
[typeOverflow.cpp:17]: (error) Memory leak: data_copy
```

```
MSVC /W4
1>  typeOverflow.cpp  nothing ☺
```

```
MSVC /analyze (PREfast)
1>  typeOverflow.cpp
bufferoverflow.cpp(13): warning : C6011:
Dereferencing NULL pointer 'data_copy'.
```

Test suites – vulnerable code, benchmark

- SAMATE Juliet Test Suite
  - huge test suite which contains at least 45000 C/C++ test cases
  - http://samate.nist.gov/SRD/testsuite.php
- Static analysis test suite for C programs
  - http://mathind.csd.auth.gr/static_analysis_test_suite/

Microsoft Zuno's bug

- No luck with Cppcheck
- No luck with PREfast
- Coverity?
- Will be solved in next lecture
  - fuzzing

```
year = ORIGINYEAR; /* = 1980 */
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

SUMMARY

Summary

- Static analysis is VERY important tool for writing secure software
  - significant portion of analysis done already by compiler (errors, warning)
- Multiple tools exists (both free and commercial)
  - Flawfinder, Cppcheck, PREfast...
  - custom rules can be written
- Static analysis cannot find all problems
  - problem of false positives/negatives
  - no substitution for extensive testing and defense programming

References

- Fortify's presentation, overview of static checking
  - http://secwg.noc.harvard.edu/archives/talks_files/chess_secure_programming.pdf
- Cppcheck presentation
  - http://www.slideshare.net/zblair/cppcheck-10316379
- Secure Programming: the Seven Pernicious Kingdoms
  - http://www.datamation.com/secu/print.php/3686291

Recommended reading

- Process of security code review
  - http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01668009
- Why cryptosystems fail, R. Anderson
  - http://www.cl.cam.ac.uk/~rja14/Papers/wcf.pdf
- Software Security Code Review
  - http://www.softwaremag.com/l.cfm?doc=2005-07/2005-07code
- Static code analysis tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
- Security in web applications (OWASP)
  - http://www.owasp.org/index.php/Code_Review_Introduction