PA193 - Secure coding principles and practices

Static analysis of source code

Łukasz Chmielewski *lukchmiel@gmail.com*
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on Cryptography and Security

Slides for comments (Thank you!)
https://drive.google.com/file/d/1--mAb7dMKd0mB6rnyToevKbmKJF8aZTQ/view?usp=sharing

PROBLEM

What is wrong with this code?

```
network_receive(uchar* in_packet, short &in_packet_len); // TLV
uchar* in = in_packet + 3;
short length = make_short(in_packet + 1);

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

memcpy(out, in, length);



network_transmit(out_packet);
```

OpenSSL Heartbleed – “packet repeater”

```
network_receive(uchar* in_packet, short &in_packet_len); // TLV
uchar* in = in_packet + 3; short length = make_short(inpacket + 1);
```

`unsigned char* in`


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

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

`unsigned char* out`


```
network_transmit(out_packet);
```

Problem?

```
network_receive(uchar* in_packet, short &in_packet_len); // TLV
uchar* in = in_packet + 3;
```

`unsigned char* in`


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

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

`in_packet_len != length + 3`

`unsigned char* out`

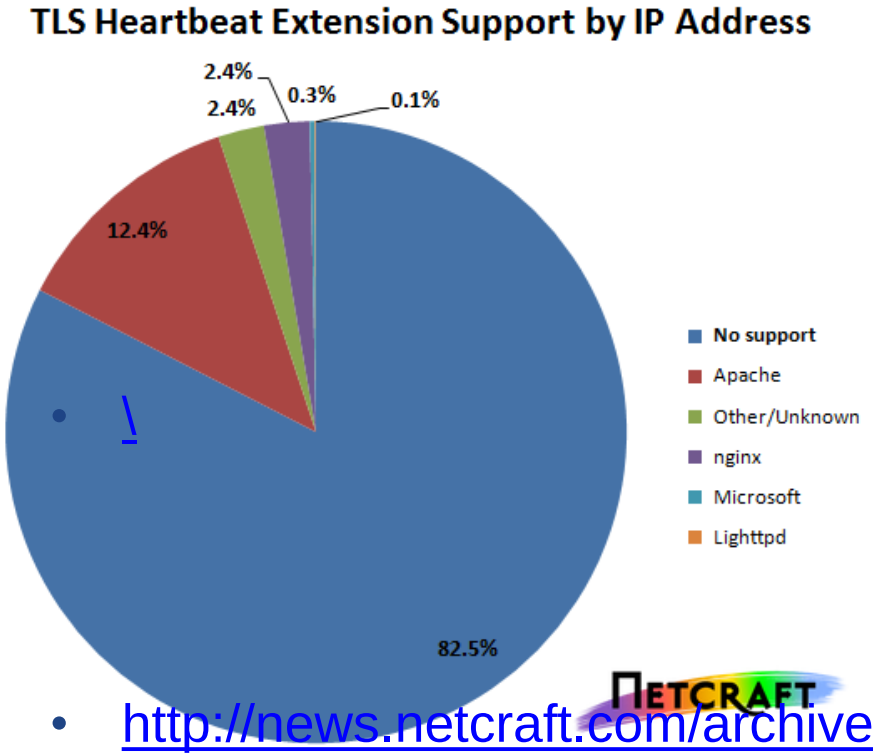

```
network_transmit(out_packet);
```

Problem!

https://heartbleed.com

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

Top questions (1)

Anonymous 0

Is information disclosure vulnerability relevant for heap and dynamically allocated memory if language has garbage collection?

Join at

slido.com

#pa193_2021

STATIC AND DYNAMIC ANALYSIS

How to find bugs in code?

- Manual analysis of code
  - code review, security code review
- Manual “dynamic” testing
  - running program, observe expected output
- Automated analysis of code without execution
  - 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
    - (outside the scope of this course, consider IA169)
- 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

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…

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

Type checking

- Type checking – performed by compiler
  - errors against language rules prevents compilation
  - warnings usually issued when problematic type manipulation occur
  - false positives possible (short=int=short), but don't ignore!
- 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
    - https://github.com/darcyliu/google-styleguide/blob/master/cppguide.xml
    - https://github.com/google/styleguide/blob/gh-pages/cpplint/cpplint.py
- Compiler warnings `gcc -Wall` `gcc -Wextra`

Bug finding

- No language errors != secure program
  - finding bugs, even when language permits it
- Examples:
  - Buffer overflow possible?
  - User input formatted into system() call?
  - Hard-coded secrets?
- Tool 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
- Difference between compiler (e.g., gcc) and additional tool (e.g., cppcheck):
  - Compiler must never report error that isn't (lang. standard)
  - Compiler must report low # of false warning (as heavily used by normal "uneducated" developers)
  - Tool executed for automatic reporting should have low # of false warnings (otherwise untrusted)
  - Tool executed during manual code review / pentest can have higher # of false warnings (as filtered by expert)

BEFORE DIGGING TO CONCRETE TOOLS…

Static analysis limitations

- Overall program architecture is not understood
  - sensitivity of program path
  - impact of errors on other parts
- Application semantics is not understood
  - Is string returned to the user? Can string also contain passwords?
- Social context is not understood
  - 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 real errors
  - 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 (for not fixing other problems reported by tool)
- False negatives
  - real errors NOT reported by a tool
  - missed problems, e.g., 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

Should you fix it even when foo() is not called?

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 as 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 in compile time and condition is now 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 executed and evaluated

BUILD-IN COMPILER ANALYSIS

Warnings – how compiler signals potential troubles

- MSVC /W n
  - /W 0 disables all warnings
  - /W 1 & /W 2 basic warnings
  - /W 3 recommended for production purposes for legacy code (default)
  - /W 4 recommended for all new compilations
  - /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

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;
}
```

int → unsigned int
-100 → 0xffffff9c

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
  - (developers may disable to suppress false warnings, missing all real ones)
- Use compiler /RTC flag

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]`**
  - http://stackoverflow.com/questions/16834588/wsign-compare-warning-in-g provides 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("Type 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

Both free and commercial tools

- Commercial tools
  - Coverity (now under Synopsys), Veracode (CA Technologies)
  - Microsoft PREfast (included in Visual Studio)
  - PC-Lint (Gimpel Software), Klocwork Insight (Perforce)
- Free tools
  - **CppCheck** http://cppcheck.sourceforge.net/
  - **Clang static analyzer** https://clang-analyzer.llvm.org/
  - **csmock** (multiple static analysers including clang, gcc, cppcheck, shellcheck, pylint, Bandit, Smatch, Coverity)
  - **SpotBugs** https://github.com/spotbugs/spotbugs (for Java programs, originally named FindBugs)
  - **PMD** https://pmd.github.io/
  - **ShellCheck** https://www.shellcheck.net/
  - Flawfinder https://www.dwheeler.com/flawfinder/, Splint http://www.splint.org/
  - Rough Auditing Tool for Security (RATS) http://code.google.com/p/rough-auditing-tool-for-security/
  - ...

Cppcheck

- A tool for static C/C++ code analysis
  - Open-source freeware, https://cppcheck.sourceforge.net/
  - Online demo https://cppcheck.sourceforge.net/demo/
- Last version 2.7 (2022-02-05)
- 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, 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

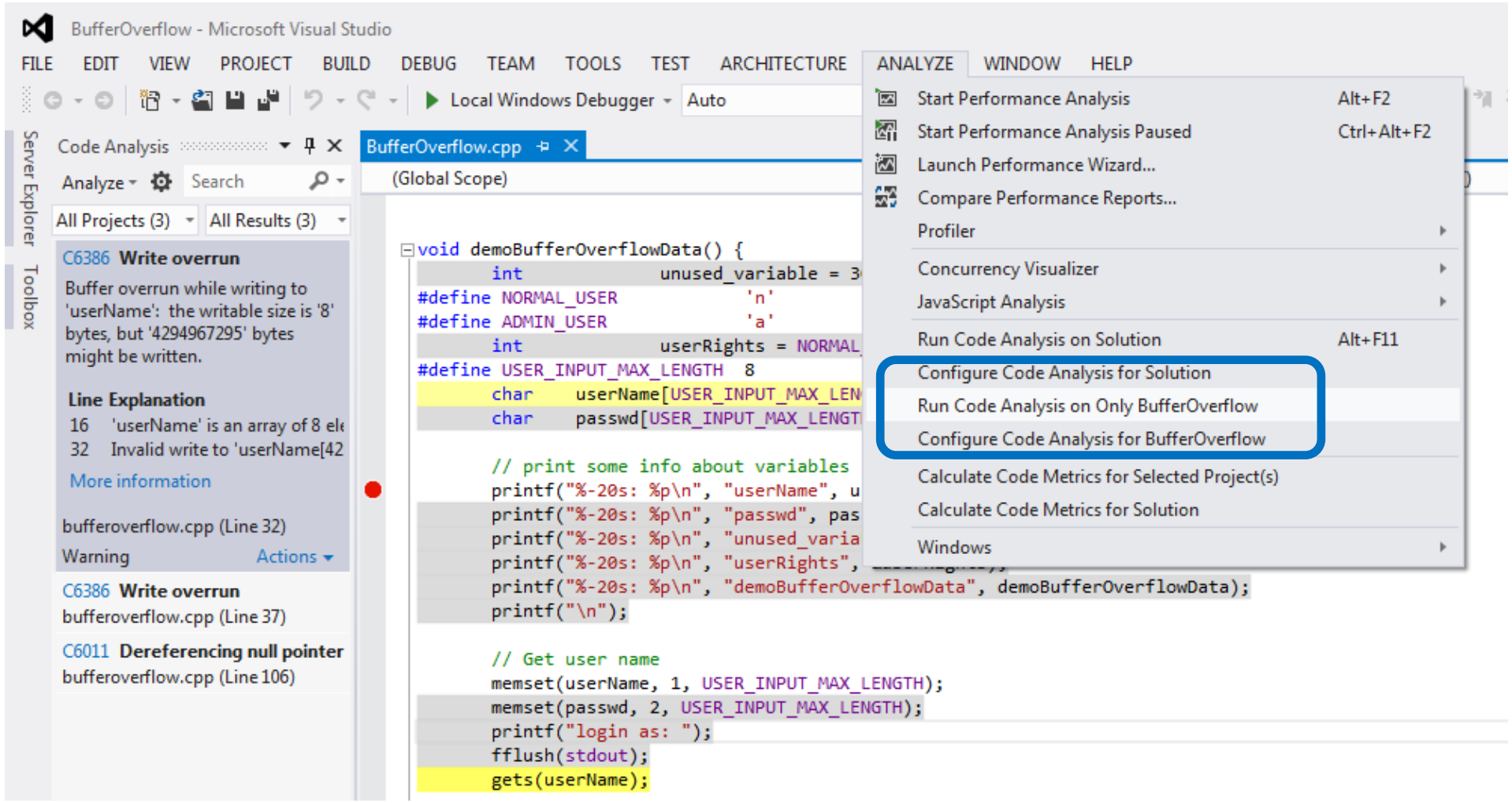
Cppcheck – complex custom rules

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

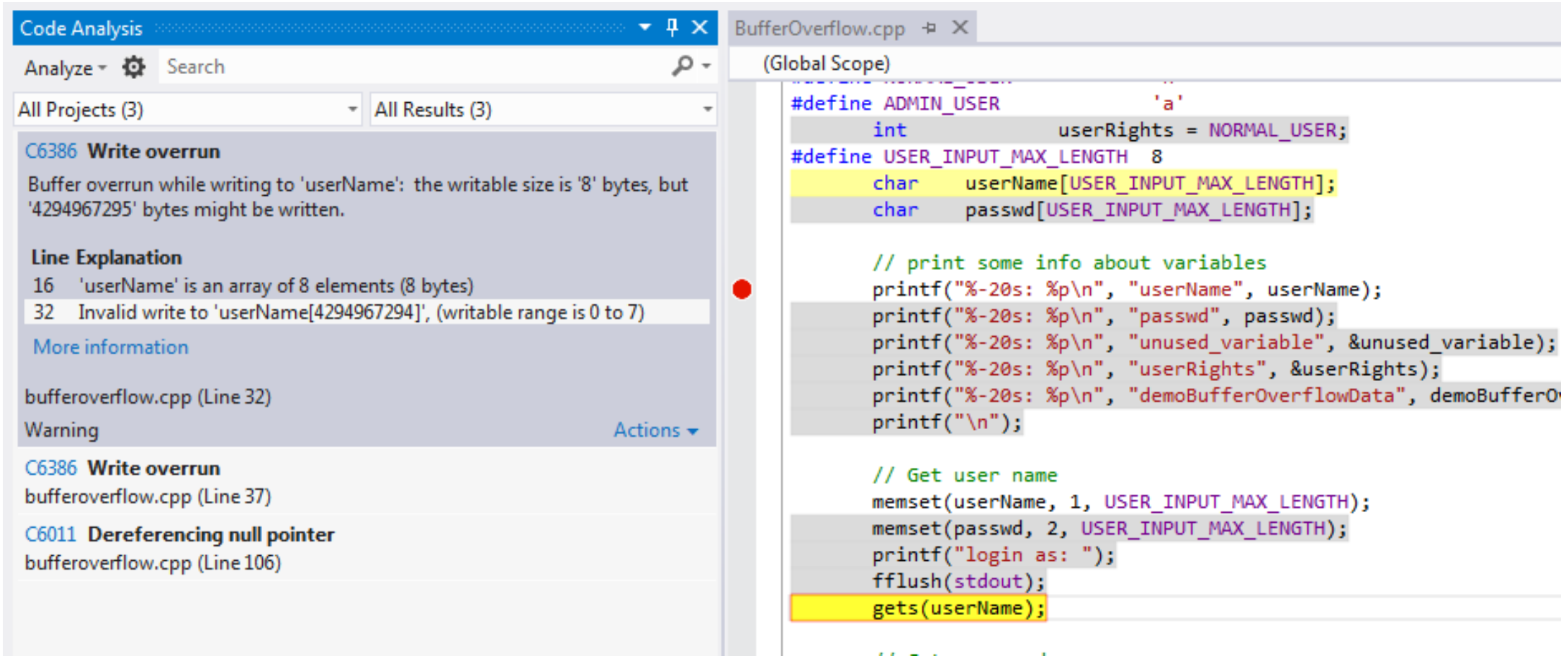
1. Use `cppcheck.exe` --debug file.cpp
   - outputs simplified code including Cppcheck's internal variable unique ID
2. Write C++ code fragment performing analysis
3. Recompile Cppcheck with new rule and execute

- Read more details
  - http://sourceforge.net/projects/cppcheck/files/Articles/
  - http://www.cs.kent.edu/~rothstei/fall_14/sec_notes/writing-rules-3.pdf

PREfast - Microsoft static analysis tool



PREfast – example bufferOverflow

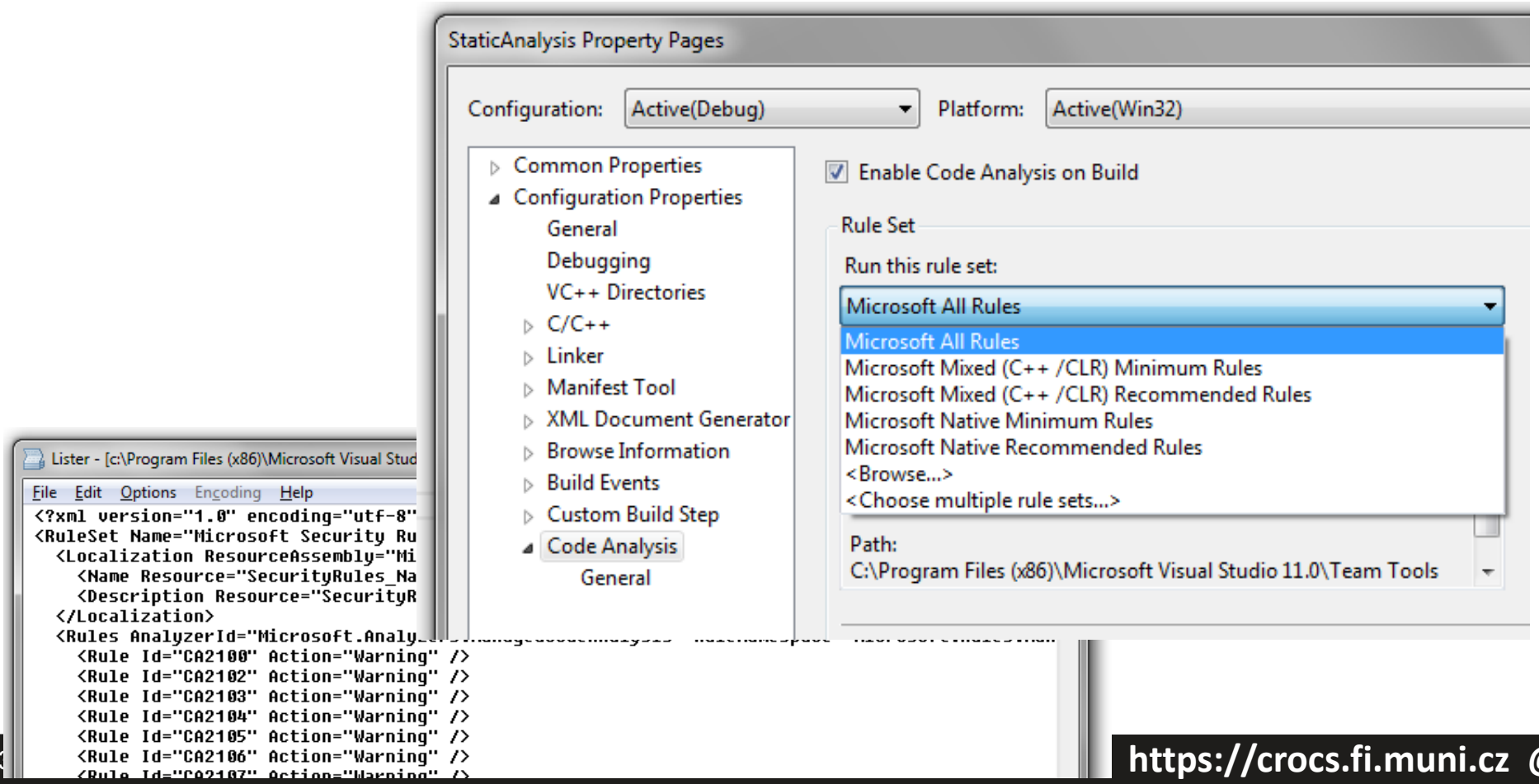


PREfast – what can be detected

- 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

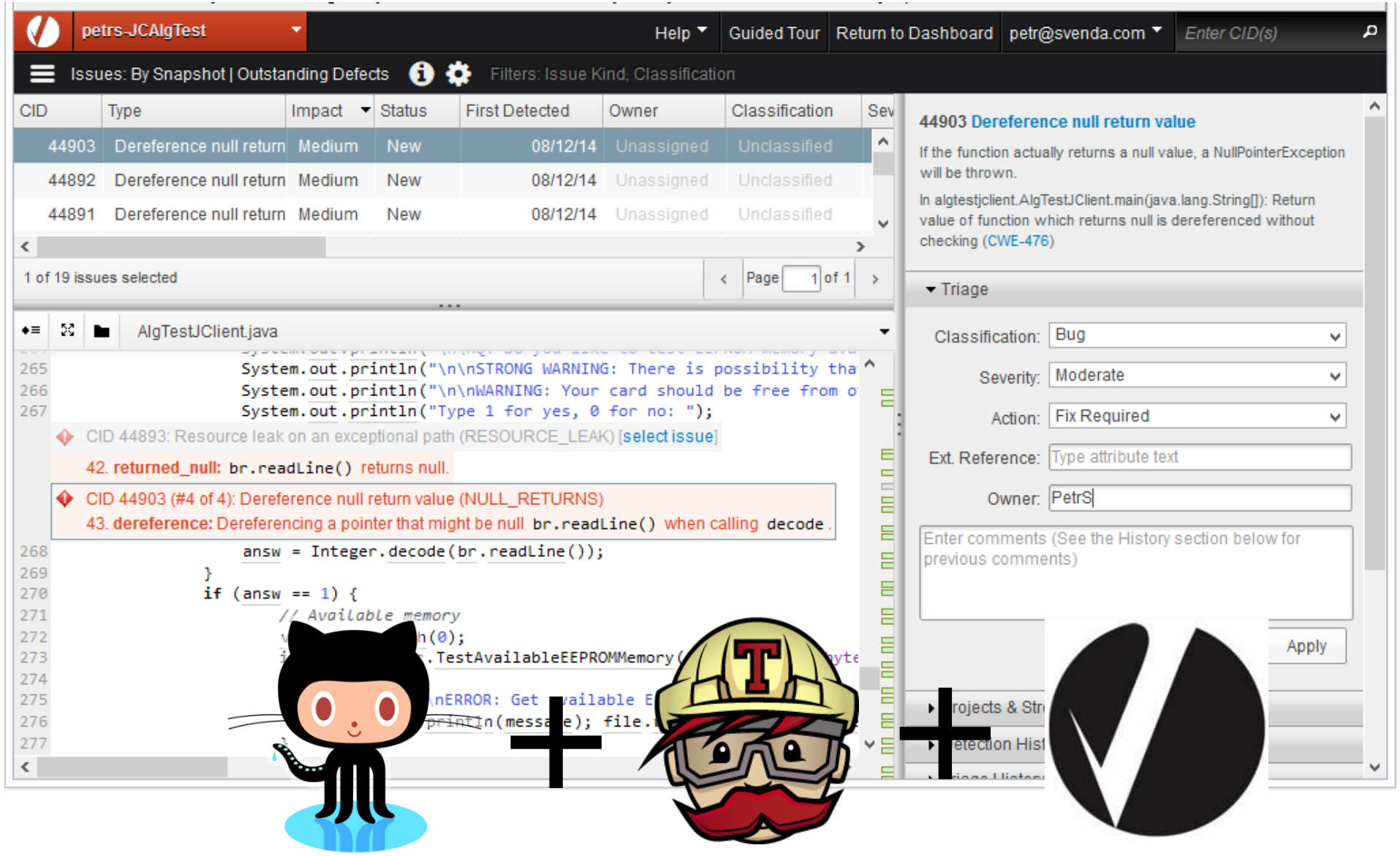
PREfast settings

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

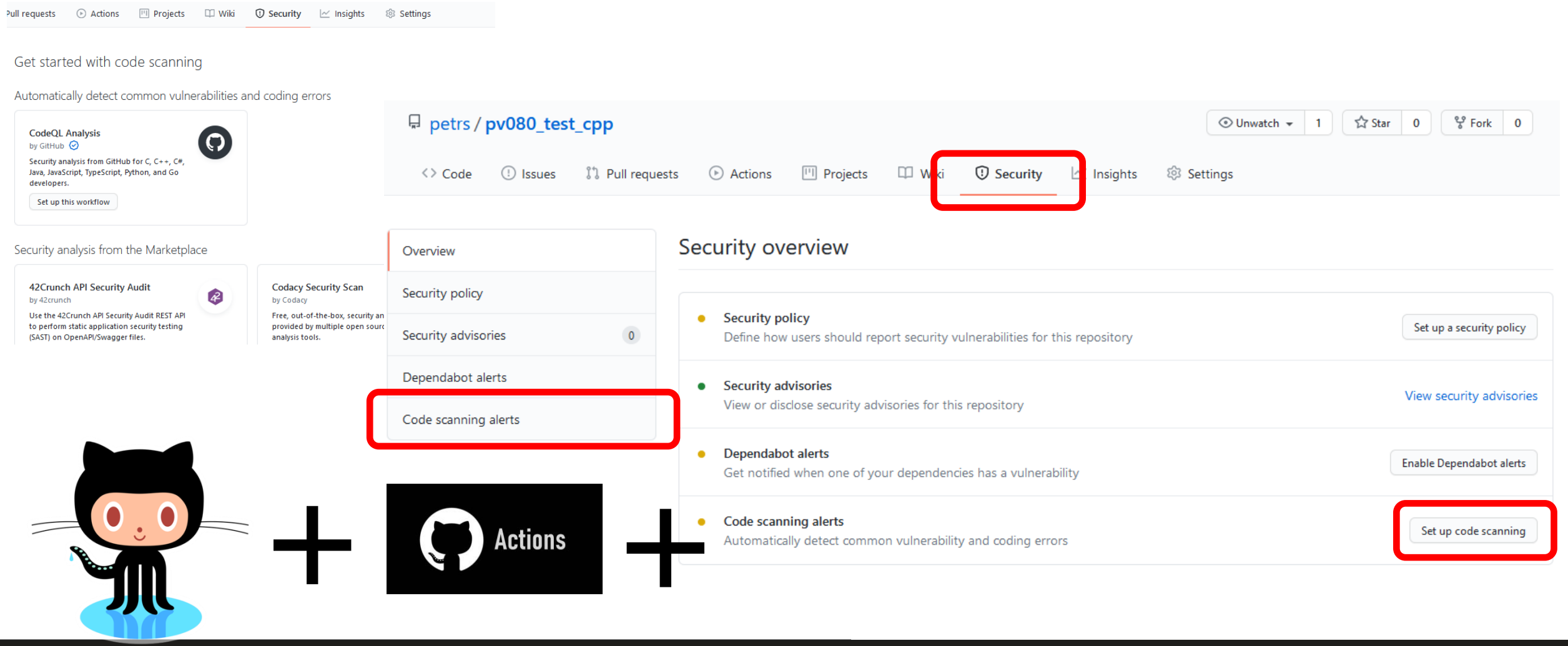


Coverity (free for open-source)

- Commercial static & dynamic analyzer
- Free for C/C++ & Java open-source projects
- https://scan.coverity.com/
- Process
  - Register at scan.coverity.com (GitHub account usage 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 (works on binary, not source)
- Can be integrated with Travis CI (continuous integration)
  - https://scan.coverity.com/travis_ci

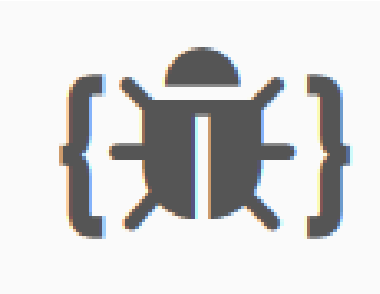

petrs-JCAlgTest
Help
Guided Tour
Return to Dashboard
petr@svenda.com
Enter CID(s)
Issues: By Snapshot | Outstanding Defects
Filters: Issue Kind, Classification
CID
Type
Impact
Status
First Detected
Owner
Classification
44903 Dereference null return Medium New 08/12/14 Unassigned Unclassified
44892 Dereference 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

Code scanning with GitHub + Actions + Codacy



SpotBugs

- Static analysis of Java programs (continuation of FindBugs)
- Extended coverage for OWASP Top 10 and CWE
- Current version 4.5.3 (2022-01-05)
  - https://github.com/spotbugs/spotbugs
  - Command-line, GUI, plugins into variety of tools
  - Support for custom rules
- FindSecurityBugs 1.11.0. (2020-10-29)
  - Additional detection rules for SpotBugs
  - https://h3xstream.github.io/find-sec-bugs/bugs.htm



PMD Source Code Analyzer

- https://pmd.github.io/
- Static analyser, mainly focused on Java, but other languages as well
- Current version 6.43.0 (2022-02-26)
- Additional features like copy-paste detector

How to reason about available tooling

- Understand problems
  - Previous ones, likely to repeat, patterns…, read bug dissection reports
- Understand principles of solution
  - What tool is used to detect problem, how was tool configured…
- Find suitable tooling for your environment
  - Language, operating system…
- Integrate, automate (CI)
  - Run tests and analysis tools frequently and automatically
- Understand limitations (what is not detected)

How many false positives are too many?

- *“Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.”*

FindBugs™ Fact Sheet

STATIC ANALYSIS IS NOT PANACEA

From 1. lecture

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 1. 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 allocation: %d\n", b
 data_copy = (some_structure*) malloc(
 if (itemPosition >= 0 && itemPosition
    memcpy(&(data_copy[itemPosition]),
 }
 else {
    printf("Out of bound assignment");
    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'.
```

From 1. lecture

What potential bug was not found?

```
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 allocation: %d\n", bytesToAllocation);
 data_copy = (some_structure*) malloc(bytesToAllocation);
 if (itemPosition >= 0 && itemPosition < totalItemsCount) {
    memcpy((&(data_copy[itemPosition]), pItem, sizeof(some_structure));
 }
 else {
    printf("Out of bound assignment");
    return;
 }
 free(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
  - https://web.archive.org/web/20110623134953/http://mathind.csd.auth.gr/static_analysis_test_suite/
- Suitable for testing new methods, but NOT for comparison of existing commercial products
  - Public suites, products already optimized for it

SUMMARY

Summary

- Static analysis is VERY important tool for writing secure software
  - Significant portion of analysis done already by compiler (errors, warnings)
  - Can run on unfinished code
- Multiple tools exist (both free and commercial)
  - Predefined set of rules, custom rules can be also written
  - Differ in capability, supported languages, target audience, maturity…
  - Experiment with available tools and find the right for your scenario
- Static analysis cannot find all problems
  - Problem of false positives/negatives
  - No substitution for extensive testing and defensive programming

Mandatory reading

- Coverity open-source reports 2013/2014/2017/2020
  - Report of analysis for open-source projects
  - https://na-sjf.marketo.com/rs/appsec/images/2013-Coverity-Scan-Report.pdf
  - https://na-sjf.marketo.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf
  - https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/SCAN-Report-2017.pdf
  - https://ttpsc.com/wp3/wp-content/uploads/2020/10/2020-ossra-report.pdf
  - https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2021.pdf
- How open-source and closed-source compare w.r.t. number of defects?
- How open-source vs. closed-source address OWASP Top 10?
- What are typical issues in C/C++ code?
- How situation changed between 2013 and 2021?

Top questions (1)

Anonymous 0

Is information disclosure vulnerability relevant for heap and dynamically allocated memory if language has garbage collection?

Questions ?

Join at

slido.com

#pa193_2021