PV286 - Secure coding principles and practices

Static analysis of source code

Łukasz Chmielewski ✉ ***chmiel@fi.muni.cz*** **(based on the lecture by P. Svenda)**
(email me with your questions/feedback)
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on
Cryptography and Security

This Lecture

- Today we cover static analysis of source code
- Some hard topics were covered too fast last time, so I will finish the lecture.
- Split version (wrt. animations) of the lecture one is also uploaded.
- Resources:
  - Recording should be available around Wednesday (but we will see).
  - An old version of the lecture (slightly shorter but well-recorded, from 2021):
    - https://is.muni.cz/auth/el/fi/jaro2022/PA193/um/video/PA193_03_StaticChecking_2022.video5
  - Last year (worse quality):
    - https://is.muni.cz/auth/el/fi/jaro2023/PV286/um/vi/136775435/
  - Materials:
    - https://is.muni.cz/auth/el/fi/jaro2024/PV286/um/

Static analysis of source code

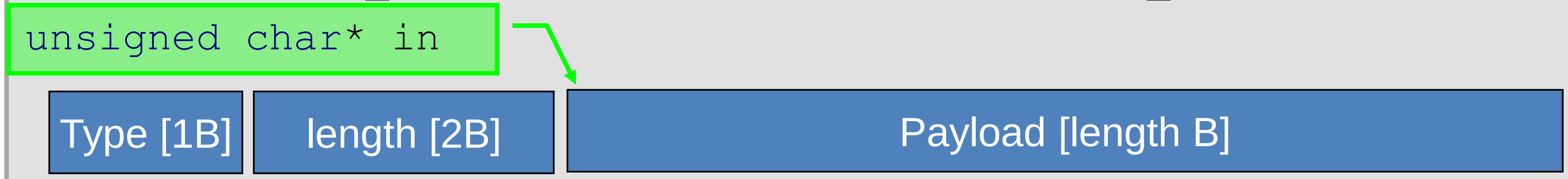
PROBLEM

What is wrong with this code?

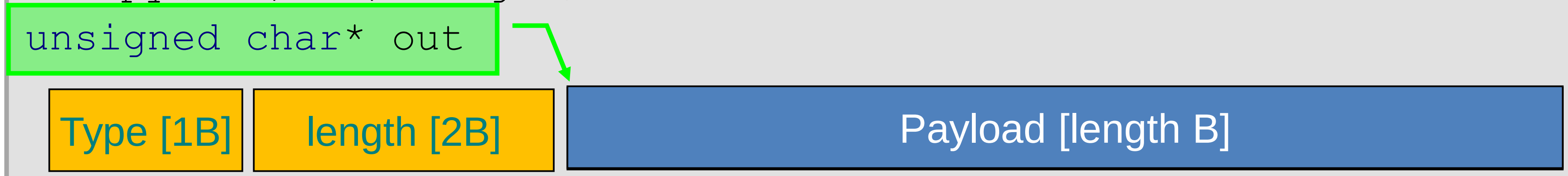
```
network_receive(uchar* in_packet, short &in_packet_len); // TLV
uchar* in = in_packet + 3;
short length = make_short(in_packet + 1);

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);
```



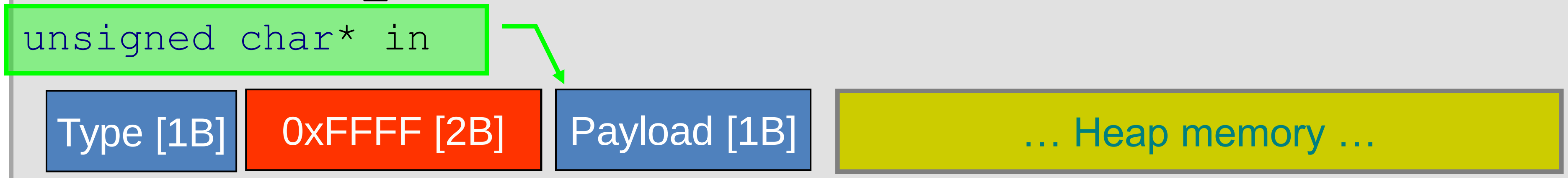
```
uchar* out_packet = malloc(1 + 2 + length);
uchar* out = out_packet + 3;

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



```
network_transmit(out_packet);
```

Problem?

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

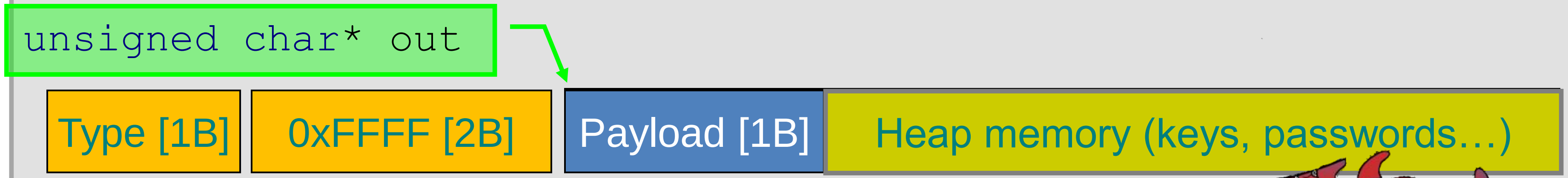


unsigned char* in

Type [1B] | 0xFFFF [2B] | Payload [1B] | … Heap memory …

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

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

in_packet_len != length + 3



unsigned char* out

Type [1B] | 0xFFFF [2B] | Payload [1B] | Heap memory (keys, passwords…)

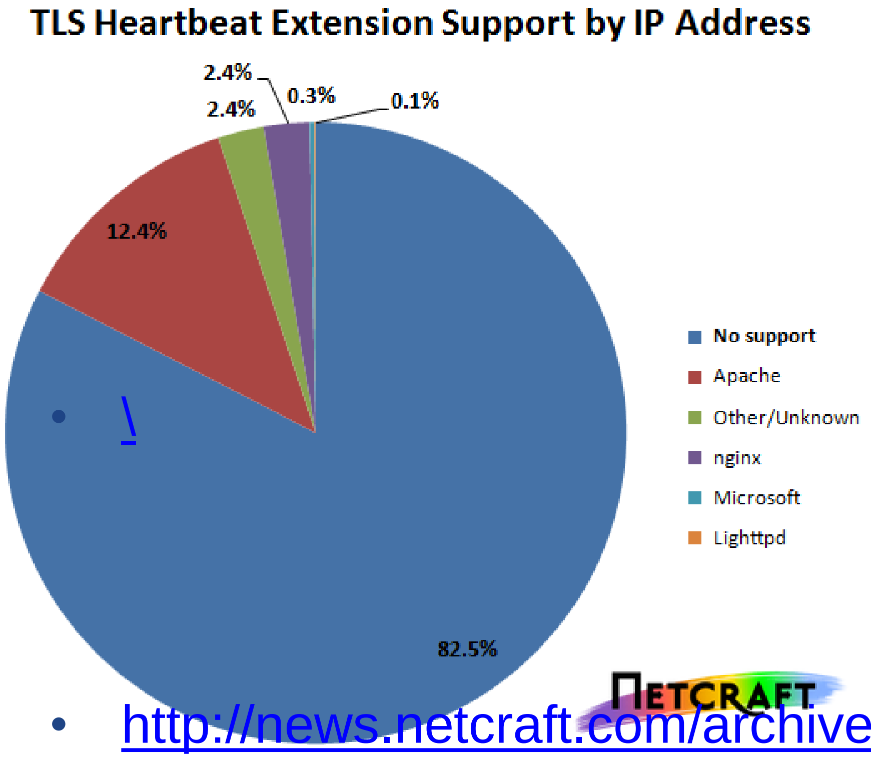
```
network_transmit(out_packet);
```

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

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)
  - But 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

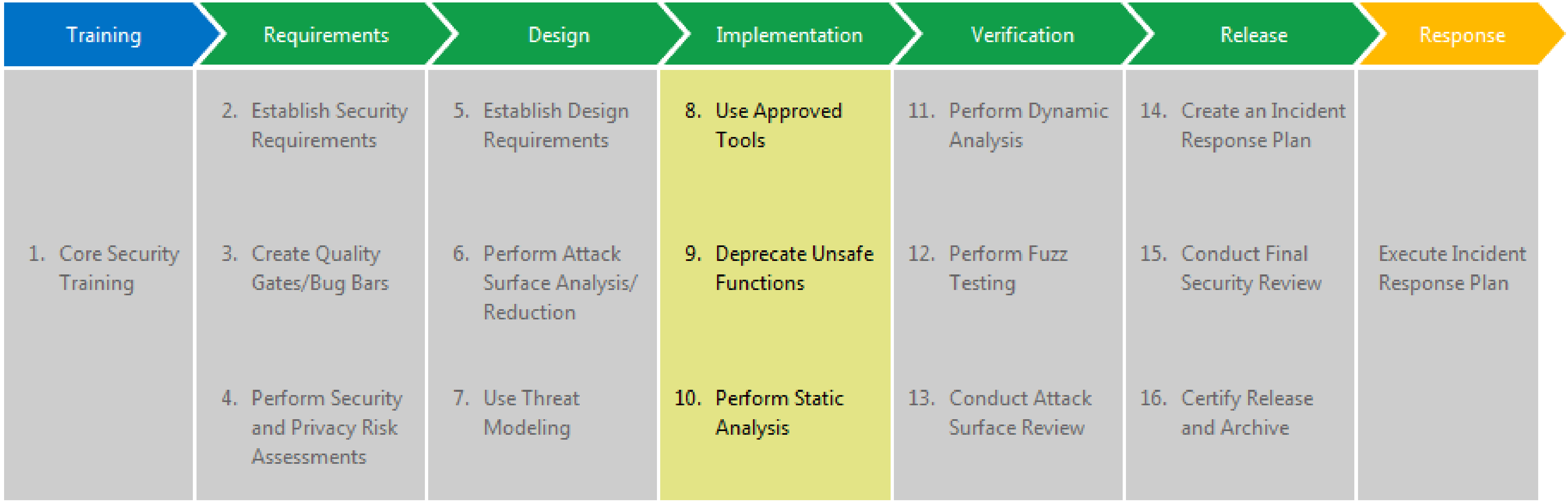
How to find bugs in code?

- Manual 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
	2. Establish Security Requirements	5. Establish Design Requirements	8. Use Approved Tools	11. Perform Dynamic Analysis	14. Create an Incident Response Plan	
1. Core Security Training	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 https://learn.microsoft.com/en-us/windows/security/threat-protection/msft-security-dev-lifecycle

Seven Touchpoints for Software Security (by Cigital)

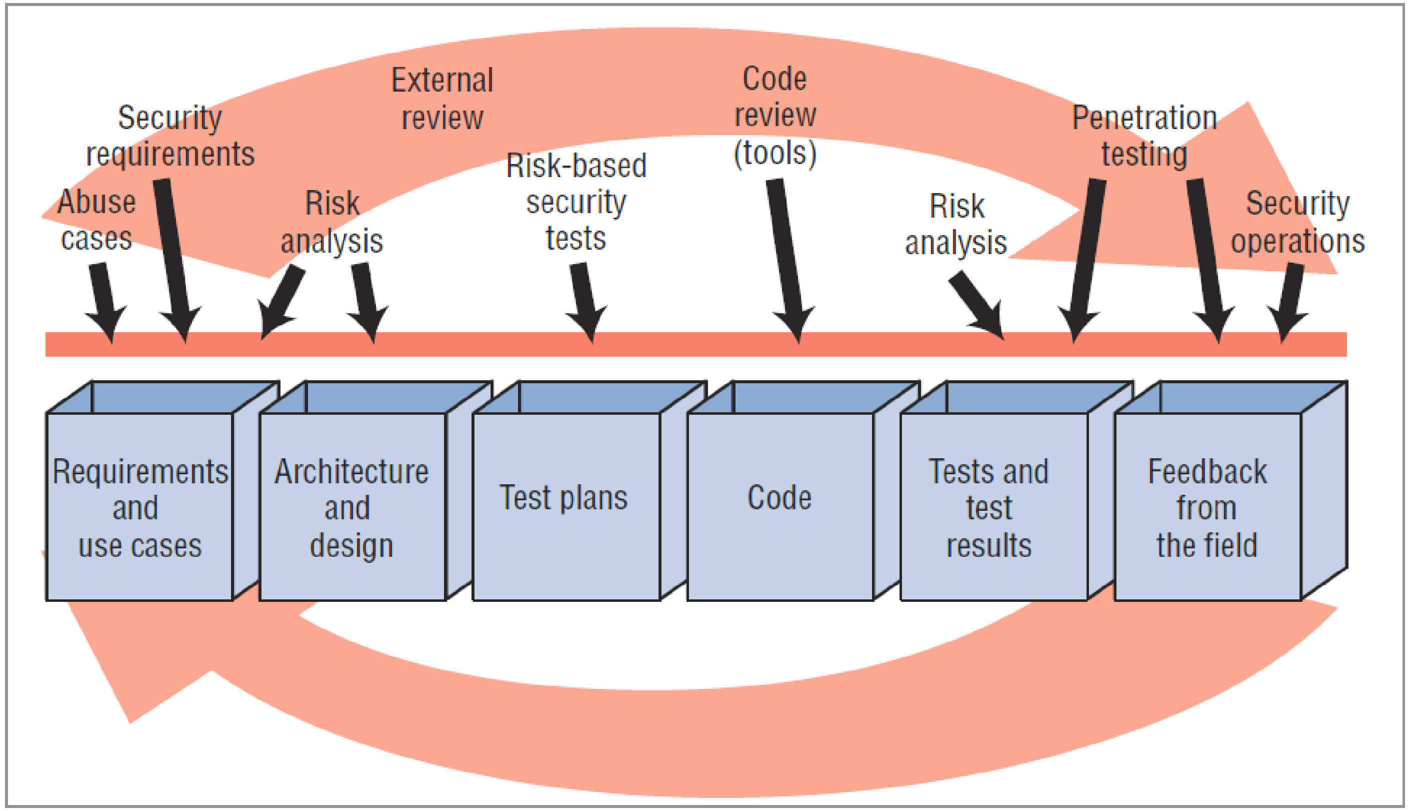


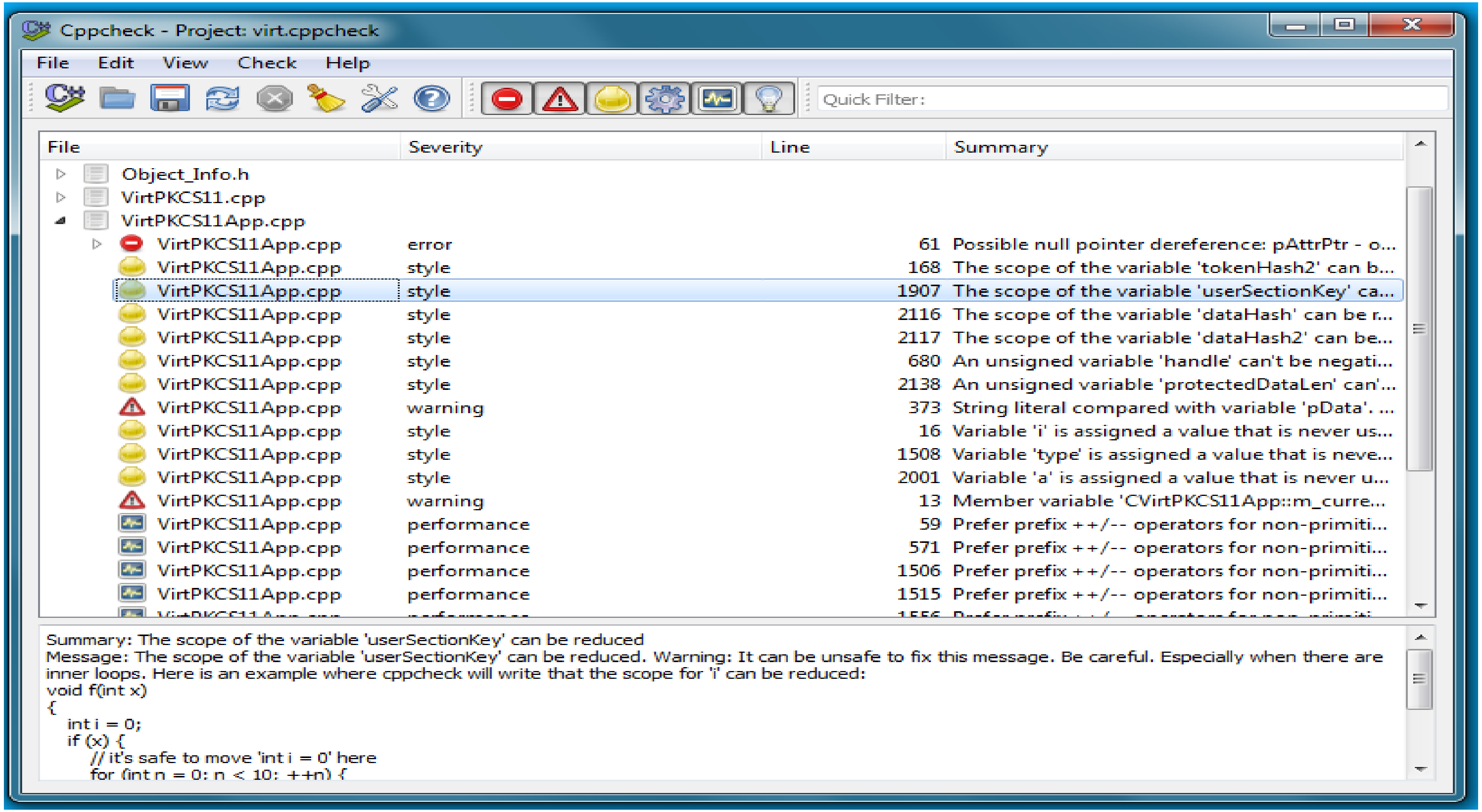
Figure 1. The Cigital Touchpoints methodology. Software security best practices (arrows) applied to various software artifacts (boxes).

http://www.swsec.com/resources/touchpoints/

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



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) {
```

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`

Program formal 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 course, consider IA169)

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 added (same as for #define)

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

```
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

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

Example (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 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

warning C4018: '>=' : signed/unsigned mismatch cont’d

But why? Rules:

- … The usual arithmetic conversions are rules that provide a mechanism to yield a common type when both operands of a binary operator are balanced to a common type or the second and third operands of the conditional operator (? :) are balanced to a common type.
- Conversions involve two operands of different types, and one or both operands may be converted. Many operators that accept arithmetic operands perform conversions using the usual arithmetic conversions. After integer promotions are performed on both operands, the following rules are applied to the promoted operands:

1. If both operands have the same type, no further conversion is needed.
2. If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
3. If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
4. …

More here: https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules#:~:text=The%20usual%20arithmetic%20conversions%20are,balanced%20to%20a%20common%20type.

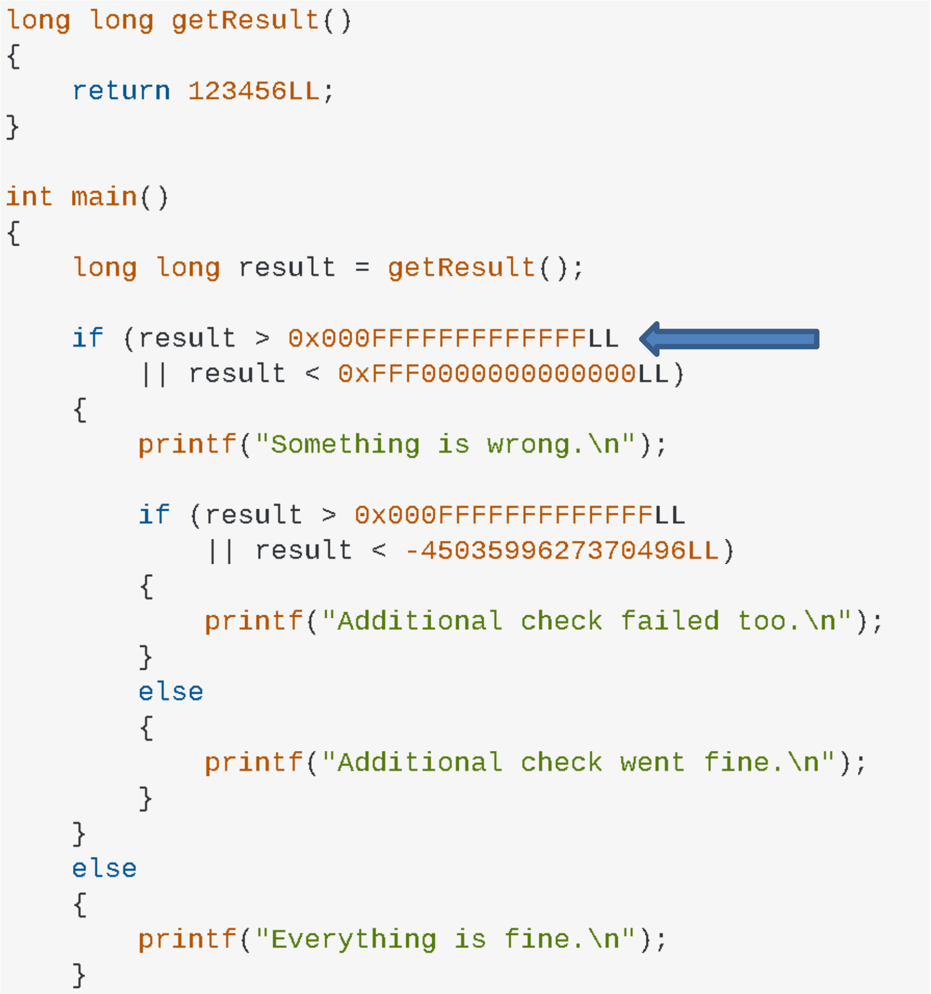
Recommendations for MSVC CL

- 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
  - Beware of also 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) ☹

Compatibility issues?



```
long long getResult()
{
    return 123456LL;
}

int main()
{
    long long result = getResult();

    if (result > 0x000FFFFFFFFFFFFFLL
        || result < 0xFFF0000000000000LL)
    {
        printf("Something is wrong.\n");

        if (result > 0x000FFFFFFFFFFFFFLL
            || result < -4503599627370496LL)
        {
            printf("Additional check failed too.\n");
        }
        else
        {
            printf("Additional check went fine.\n");
        }
    }
    else
    {
        printf("Everything is fine.\n");
    }
```

- This listing provides an example of a real problem.
- g++:

```
Something is wrong.
Additional check went fine.
```

- MSVC: `Everything is fine.`
- Why?
  - `unsigned long long` (for g++) vs `long long` (for MSVC)
  - Compatibility reasons for MSVC?

For more see http://stackoverflow.com/questions/16834588/wsign-compare-warning-in-g

Try this at home!

GCC -ftrapv

```
/* 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 analyzers 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 2.13 (2023-12-23)
- 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`**

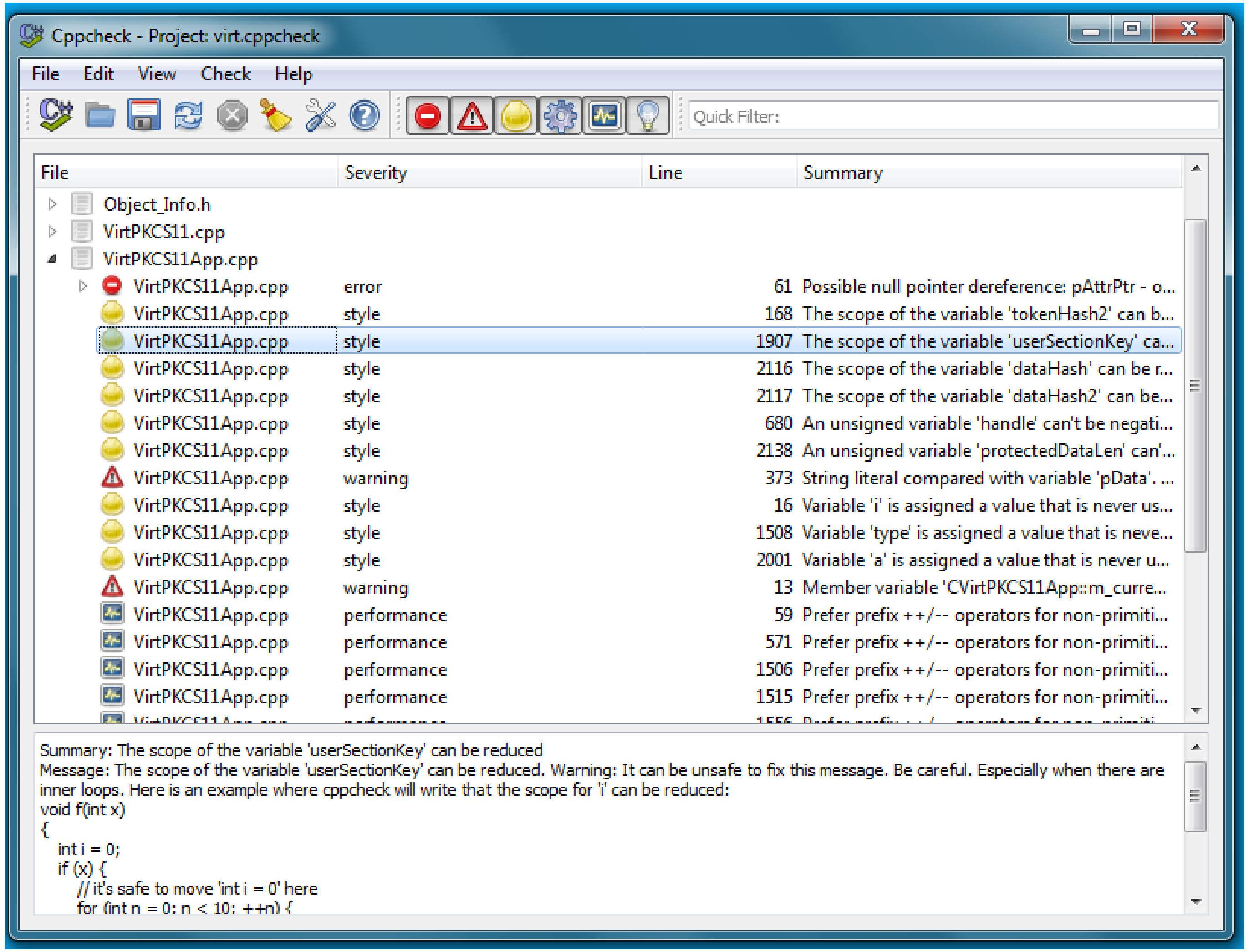
Cppcheck – what is checked?

- Bound checking for array overruns
- Suspicious patterns for class
- Exceptions safety
- Memory leaks
- Obsolete functions
- sizeof() related problems
- String format problems...
- See full list: https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/

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

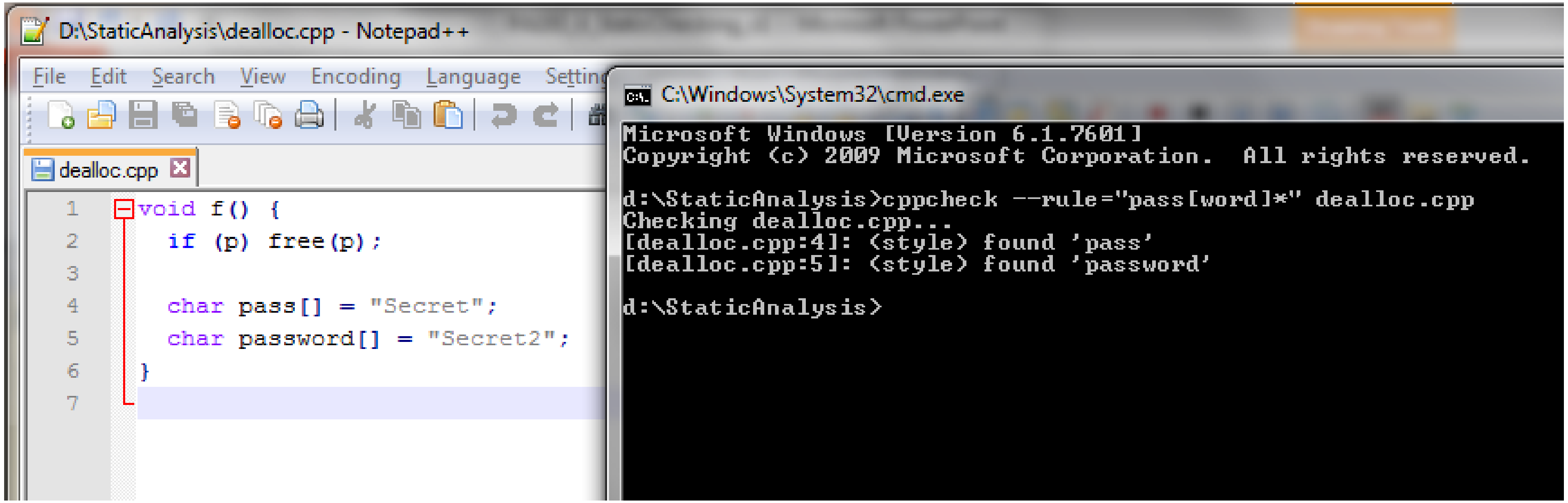


File	Severity	Line	Summary
Object_Info.h			
VirtPKCS11.cpp			
VirtPKCS11App.cpp			
VirtPKCS11App.cpp	error	61	Possible null pointer dereference: pAttrPtr - o...
VirtPKCS11App.cpp	style	168	The scope of the variable 'tokenHash2' can b...
VirtPKCS11App.cpp	style	1907	The scope of the variable 'userSectionKey' 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...
VirtPKCS11App.cpp	performance	1556	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.exe --rule="pass[word]*" file.cpp


D:\StaticAnalysis\dealloc.cpp - Notepad++
File
Edit
Search
View
Encoding
Language
Setting
dealloc.cpp
C:\Windows\System32\cmd.exe

```cpp
void f() {
   if (p) free(p);

   char pass[] = "Secret";
   char password[] = "Secret2";
}

```

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

d:\StaticAnalysis>cppcheck --rule="pass[word]*" dealloc.cpp
Checking dealloc.cpp...
[dealloc.cpp:4]: (style) found 'pass'
[dealloc.cpp:5]: (style) found 'password'

d:\StaticAnalysis>
```

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

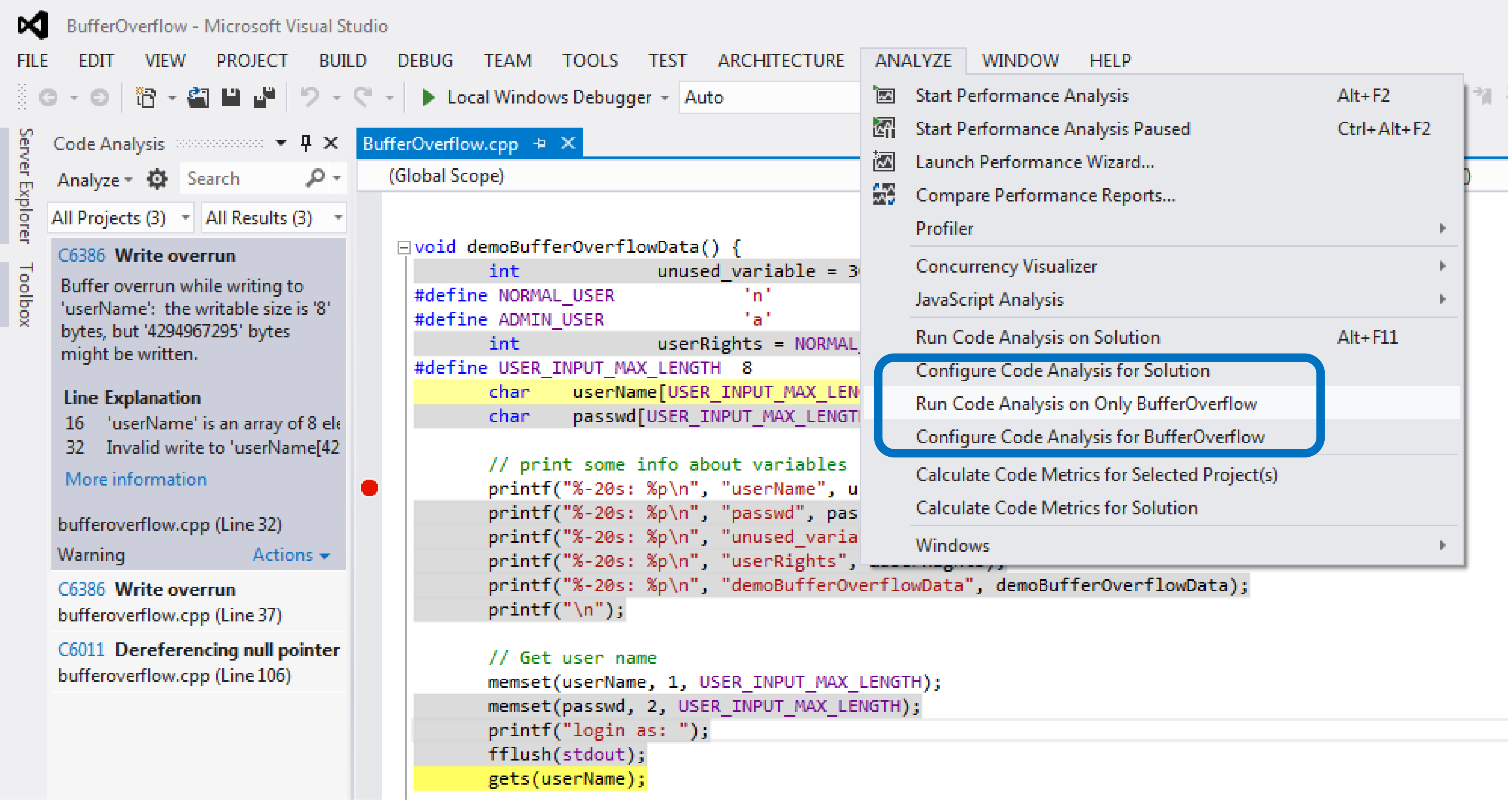
Cppcheck – complex custom rules

- Simple rules: regular expressions
- 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


BufferOverflow - Microsoft Visual Studio
FILE
EDIT
VIEW
PROJECT
BUILD
DEBUG
TEAM
TOOLS
TEST
ARCHITECTURE
ANALYZE
WINDOW
HELP
Local Windows Debugger
Auto
Start Performance Analysis Alt+F2
Start Performance Analysis Paused Ctrl+Alt+F2
Launch Performance Wizard...
Compare Performance Reports...
Profiler
Concurrency Visualizer
JavaScript Analysis
Run Code Analysis on Solution Alt+F11
Configure Code Analysis for Solution
Run Code Analysis on Only BufferOverflow
Configure Code Analysis for BufferOverflow
Calculate Code Metrics for Selected Project(s)
Calculate Code Metrics for Solution
Windows
Server Explorer
Toolbox
Code Analysis
Analyze
Search
All Projects (3)
All Results (3)
BufferOverflow.cpp
(Global Scope)
More information
Actions

C6386 **Write overrun**

Buffer overrun while writing to 'userName': the writable size is '8' bytes, but '4294967295' bytes might be written.

Line Explanation

16 'userName' is an array of 8 ele

32 Invalid write to 'userName[42

bufferoverflow.cpp (Line 32)

Warning

C6386 **Write overrun**

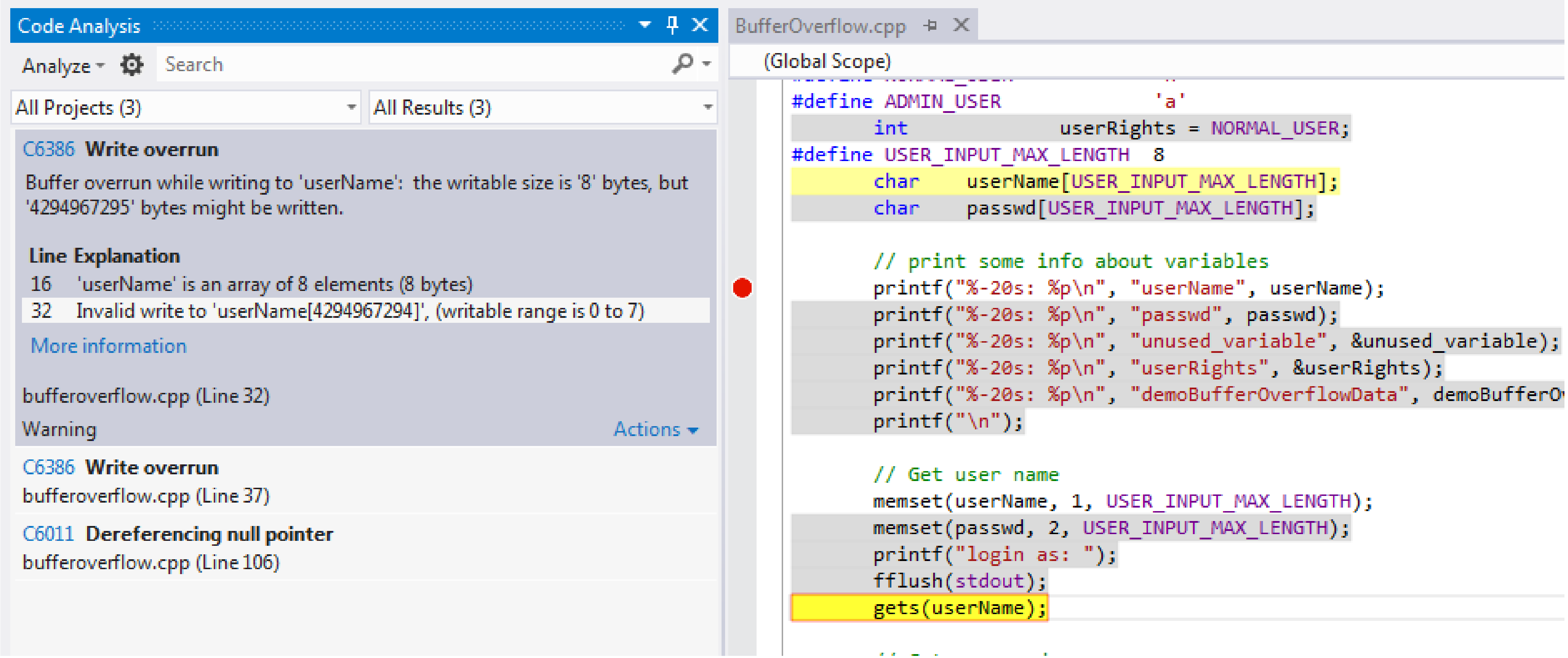
bufferoverflow.cpp (Line 37)

C6011 **Dereferencing null pointer**

bufferoverflow.cpp (Line 106)

```
void demoBufferOverflowData() {
        int                 unused_variable = 3
#define NORMAL_USER                 'n'
#define ADMIN_USER                  'a'
        int                 userRights = NORMAL_
#define USER_INPUT_MAX_LENGTH  8
        char     userName[USER_INPUT_MAX_LEN
        char     passwd[USER_INPUT_MAX_LENGT

        // print some info about variables
        printf("%-20s: %p\n", "userName", u
        printf("%-20s: %p\n", "passwd", pas
        printf("%-20s: %p\n", "unused_varia
        printf("%-20s: %p\n", "userRights",
        printf("%-20s: %p\n", "demoBufferOverflowData", demoBufferOverflowData);
        printf("\n");

        // Get user name
        memset(userName, 1, USER_INPUT_MAX_LENGTH);
        memset(passwd, 2, USER_INPUT_MAX_LENGTH);
        printf("login as: ");
        fflush(stdout);
        gets(userName);
```

PREfast – example bufferOverflow



C6386 Write overrun

Buffer overrun while writing to 'userName': the writable size is '8' bytes, but '4294967295' bytes might be written.

Line	Explanation
16	'userName' is an array of 8 elements (8 bytes)
32	Invalid write to 'userName[4294967294]', (writable range is 0 to 7)

More information

bufferoverflow.cpp (Line 32)

Warning Actions

C6386 Write overrun
bufferoverflow.cpp (Line 37)

C6011 Dereferencing null pointer
bufferoverflow.cpp (Line 106)

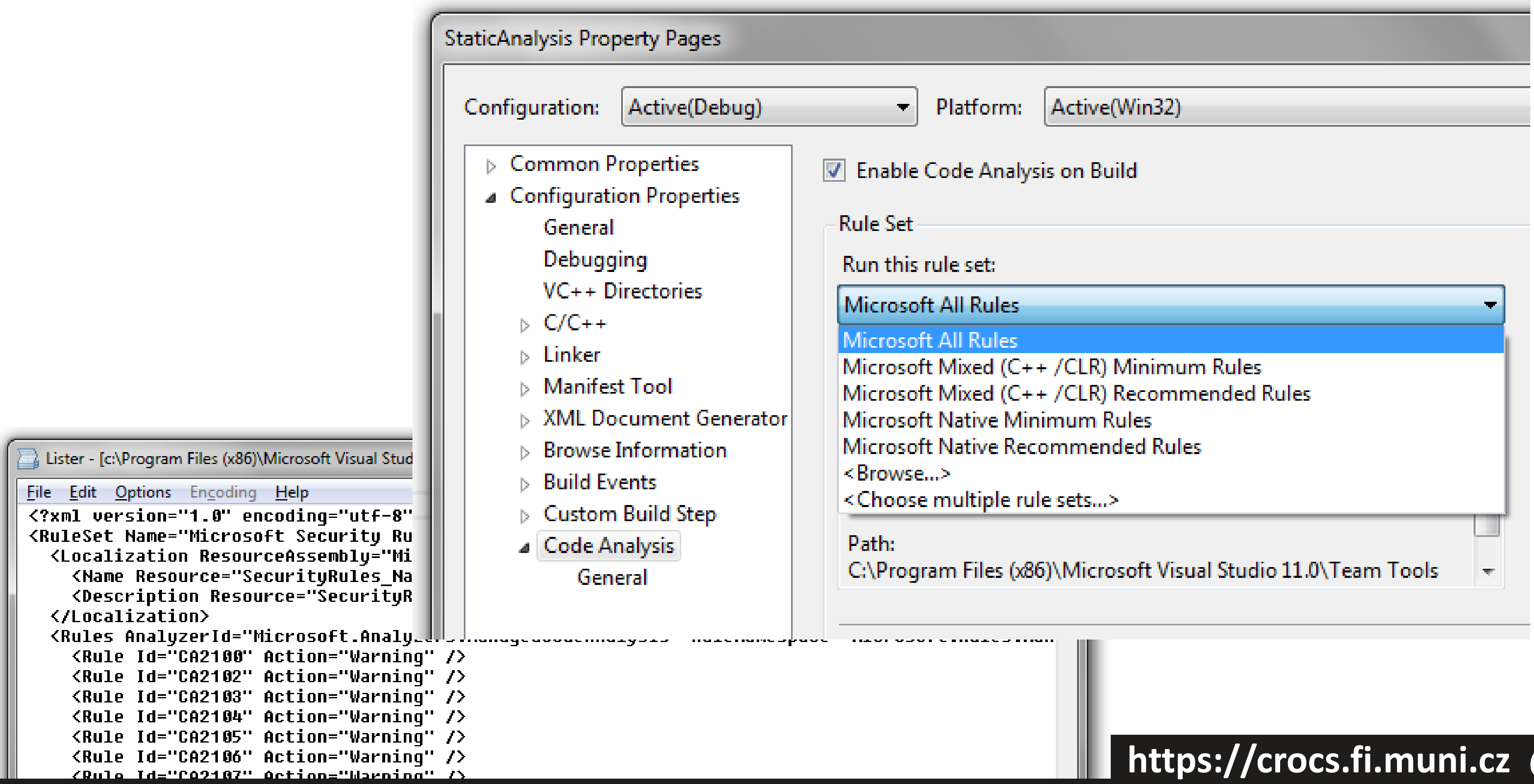
```
#define ADMIN_USER              'a'
    int             userRights = NORMAL_USER;
#define USER_INPUT_MAX_LENGTH  8
    char    userName[USER_INPUT_MAX_LENGTH];
    char    passwd[USER_INPUT_MAX_LENGTH];

    // print some info about variables
    printf("%-20s: %p\n", "userName", userName);
    printf("%-20s: %p\n", "passwd", passwd);
    printf("%-20s: %p\n", "unused_variable", &unused_variable);
    printf("%-20s: %p\n", "userRights", &userRights);
    printf("%-20s: %p\n", "demoBufferOverflowData", demoBufferO
    printf("\n");

    // Get user name
    memset(userName, 1, USER_INPUT_MAX_LENGTH);
    memset(passwd, 2, USER_INPUT_MAX_LENGTH);
    printf("login as: ");
    fflush(stdout);
    gets(userName);
```

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



StaticAnalysis Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)

☑ Enable Code Analysis on Build

Rule Set

Run this rule set: Microsoft All Rules

Microsoft All Rules
Microsoft Mixed (C++ /CLR) Minimum Rules
Microsoft Mixed (C++ /CLR) Recommended Rules
Microsoft Native Minimum Rules
Microsoft Native Recommended Rules
<Browse...>
<Choose multiple rule sets...>

Path:
C:\Program Files (x86)\Microsoft Visual Studio 11.0\Team Tools

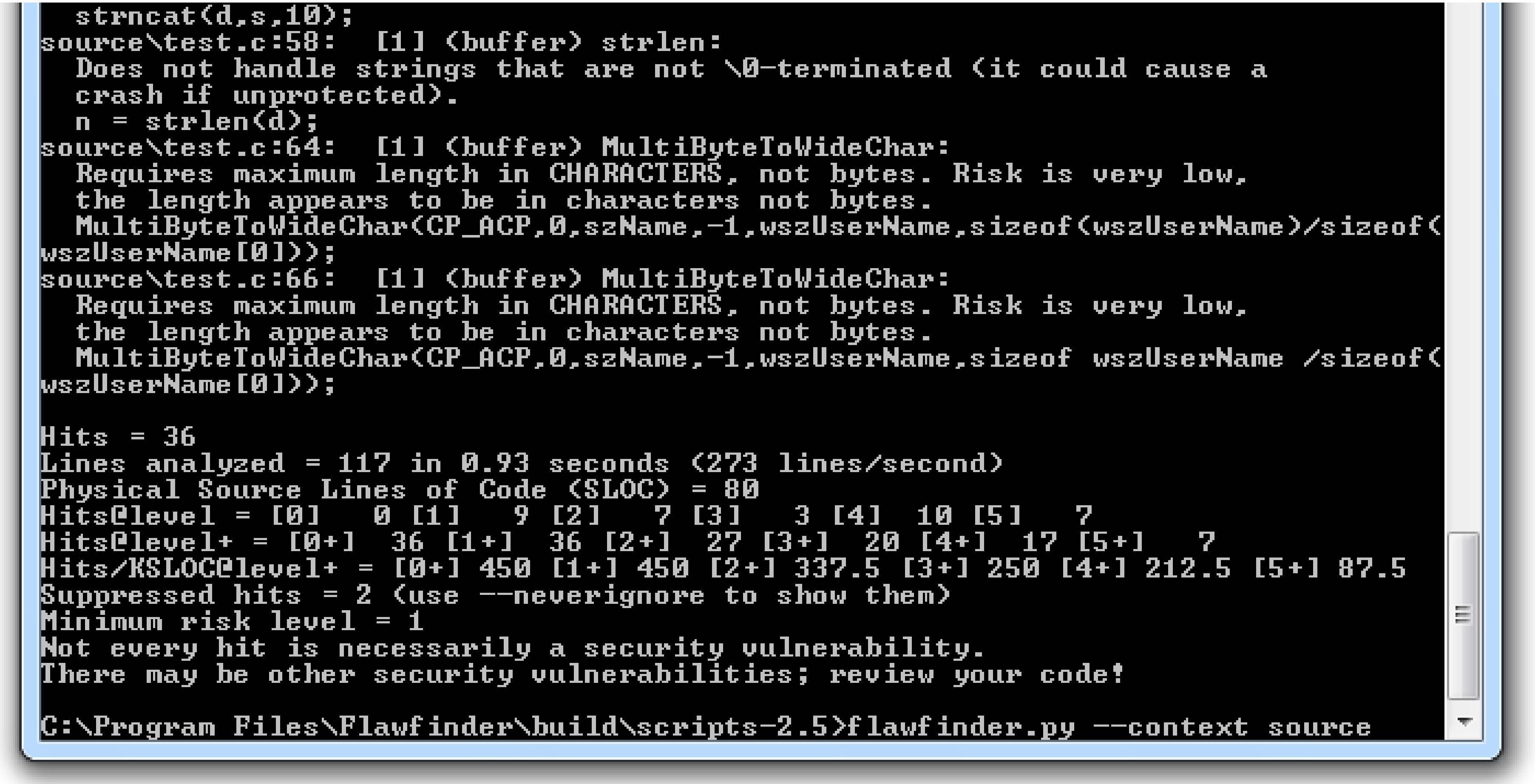
Lister - [c:\Program Files (x86)\Microsoft Visual Stud

```
<?xml version="1.0" encoding="utf-8"
<RuleSet Name="Microsoft Security Ru
  <Localization ResourceAssembly="Mi
    <Name Resource="SecurityRules_Na
    <Description Resource="SecurityR
  </Localization>
  <Rules AnalyzerId="Microsoft.Analy
    <Rule Id="CA2100" Action="Warning" />
    <Rule Id="CA2102" Action="Warning" />
    <Rule Id="CA2103" Action="Warning" />
    <Rule Id="CA2104" Action="Warning" />
    <Rule Id="CA2105" Action="Warning" />
    <Rule Id="CA2106" Action="Warning" />
    <Rule Id="CA2107" Action="Warning" />
```

Flawfinder

- Last version 2.0.19 (2021-08-29)
- 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`

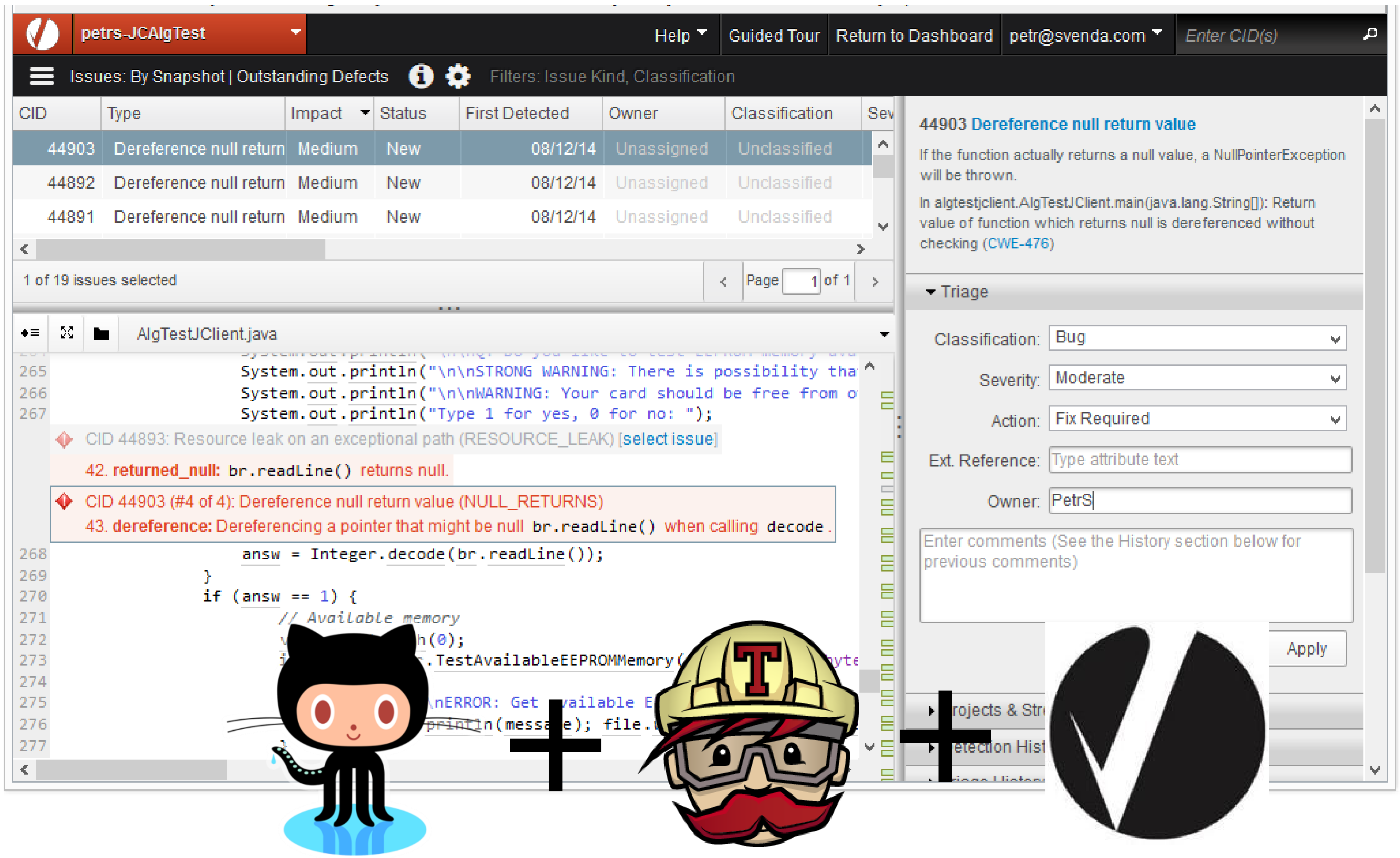
Flawfinder - example



```
  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
```

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	Sev
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

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.
                answw = Integer.decode(br.readLine());
            }
            if (answ == 1) {
                // Available memory
                    h(0);
                    .TestAvailableEEPROMMemory(                    yte
                    nERROR: Get   vailable E
                    println(messa e); file.
```

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

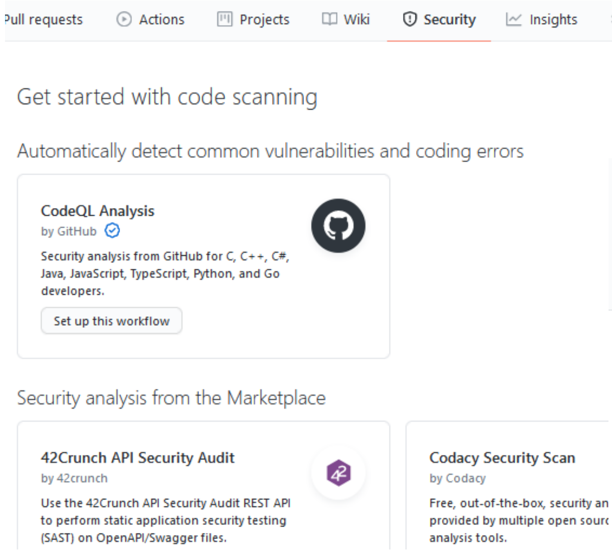
Ext. Reference: Type attribute text

Owner: PetrS

Enter comments (See the History section below for previous comments)

+ +

Code scanning with GitHub + Actions + Codacy



Get started with code scanning

Automatically detect common vulnerabilities and coding errors

CodeQL Analysis
by GitHub

Security analysis from GitHub for C, C++, C#, Java, JavaScript, TypeScript, Python, and Go developers.

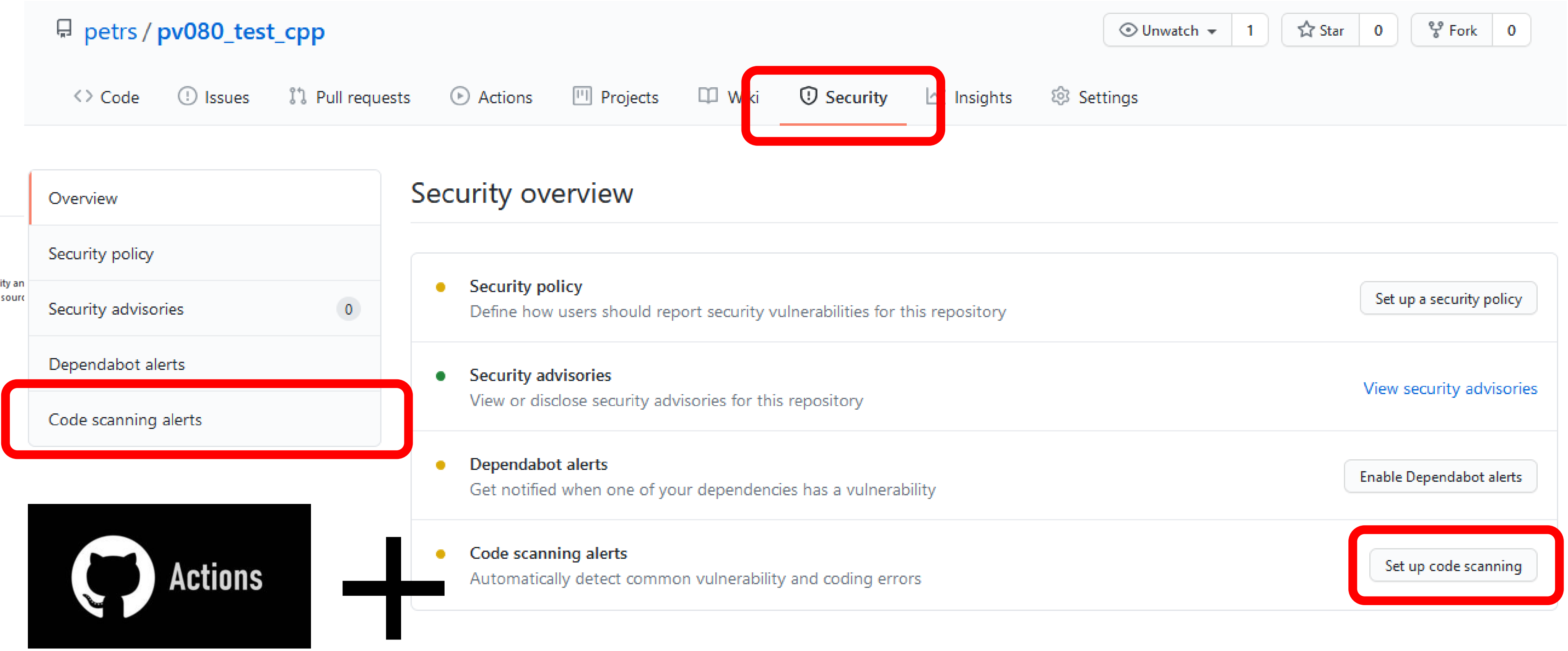
Security analysis from the Marketplace

42Crunch API Security Audit
by 42crunch

Use the 42Crunch API Security Audit REST API to perform static application security testing (SAST) on OpenAPI/Swagger files.

Codacy Security Scan
by Codacy

Free, out-of-the-box, security an provided by multiple open sourc analysis tools.



petrs / pv080_test_cpp

Security overview

Security policy
Define how users should report security vulnerabilities for this repository

Security advisories
View or disclose security advisories for this repository

Dependabot alerts
Get notified when one of your dependencies has a vulnerability

Code scanning alerts
Automatically detect common vulnerability and coding errors

+ Actions +

SpotBugs

- Static analysis of Java programs (continuation of FindBugs)
- Extended coverage for OWASP Top 10 and CWE
- Current version 4.8.3 (2023-12-1)
  - https://github.com/spotbugs/spotbugs
  - Command-line, GUI, plugins into variety of tools
  - Support for custom rules
- FindSecurityBugs 1.12.0. (2022-04-06)
  - 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 7.0.0-rc4 (30-September-2023)
- 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://ieeexplore.ieee.org/document/6032220
- 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 2013/2014/2017/2020/2021
  - Report of analysis for open-source projects
  - https://web.archive.org/web/20200320234505/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://web.archive.org/web/20220315064102/https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2021.pdf
  - https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf
  - https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2023.pdf
- How open-source and closed-source compare w.r.t. number of defects?
- How does open-source vs. closed-source address OWASP Top 10?
- What are typical issues in C/C++ code?
- How has the situation changed from 2017 onward?
- Optional reading: https://www.cl.cam.ac.uk/~rja14/Papers/wcf.pdf

Questions