

PA193 - Secure coding principles and practices



Static analysis of source code

Petr Švenda  svenda@fi.muni.cz  [@rngsec](https://twitter.com/rngsec)
Centre for Research on Cryptography and Security, Masaryk University

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 Heartbeat – “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
```

Type [1B]

length [2B]

Payload [length B]

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

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

```
unsigned char* out
```

Type [1B]

length [2B]

Payload [length B]

```
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!



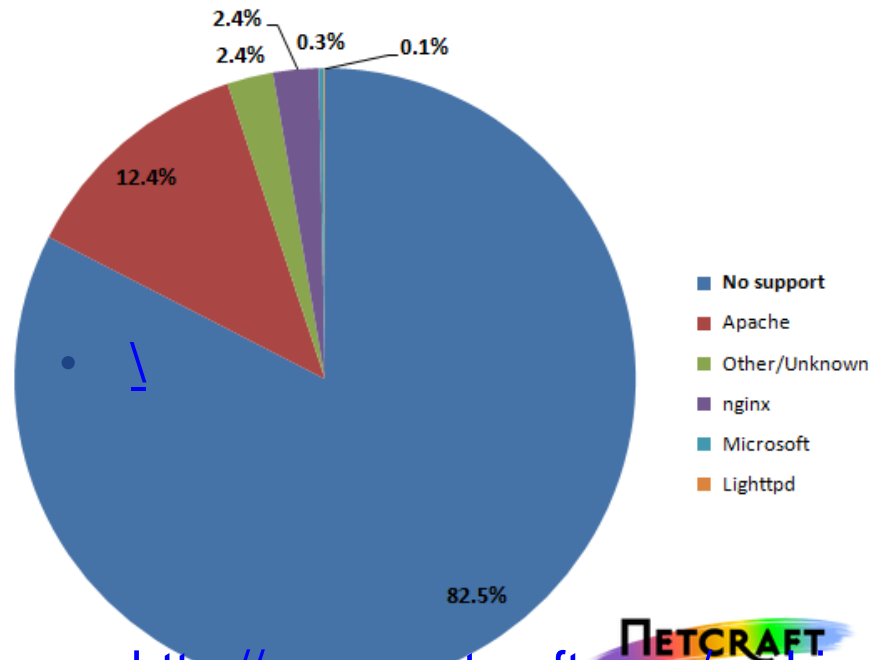
How serious the bug was?

17% SSL web servers (OpenSSL 1.0.1)

[Twitter](#), [GitHub](#), [Yahoo](#), [Tumblr](#), [Steam](#), [DropBox](#), [DuckDuckGo](#) ...
<https://seznam.cz>, <https://fi.muni.cz> ...



TLS Heartbeat Extension Support by IP Address



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





Anonymous

0 

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

- **Place/upvote questions in slido while listening to lecture video**
- **We will together discuss these during every week lecture Q&A (every Monday, 11-12:00)**

Join at
slido.com
#pa193_2021



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 compilation
 - static analysis (pattern matching, symbolic execution)
- Automated analysis of code with execution
 - dynamic analysis (running code)
- Automated testing of inputs (fuzzing)

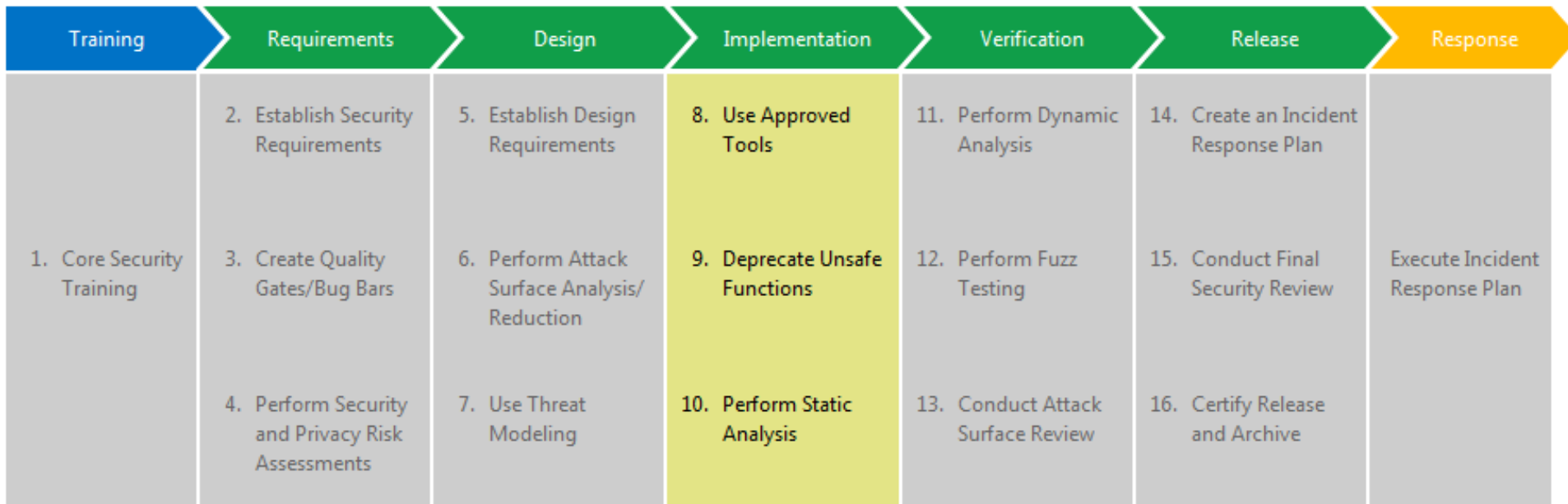


Approaches for automated code review

- **Formal methods** (mathematical verification)
 - requires mathematical model and assertions
 - often requires modeling the system as finite state machine
 - verification of every state and transition
 - (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



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



Cigital Touchpoints methodology

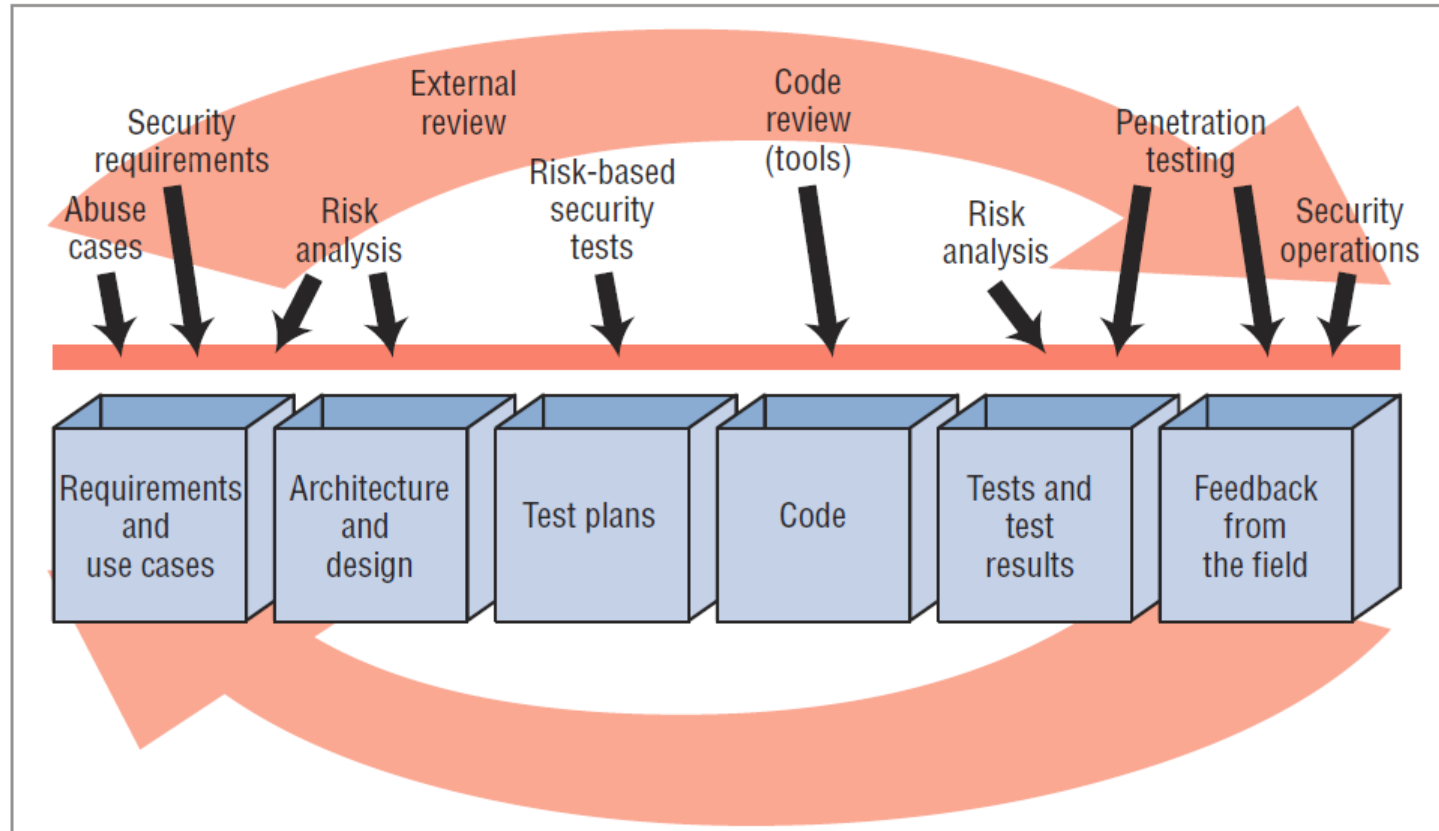


Figure 1. The Cigital Touchpoints methodology. Software security best practices (arrows) applied to various software artifacts (boxes).



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

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-prim...
VirtPKCS11App.cpp	performance	1556	Prefer prefix ++/-- operators for non-prim...

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



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 through 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 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
 - <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 course, consider IA169)

Bug finding

- No language errors \neq 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 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 $x + y \neq 2$ 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 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 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;
}
```

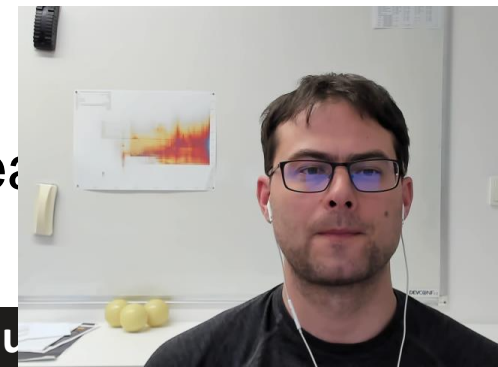


int → unsigned int
-100 → 0xffffffff9c



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 \rightarrow floating point, signed \rightarrow unsigned, ...
- GCC `-Wsign-compare`
 - signed \rightarrow 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) ☹️



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-n>



STATIC ANALYSIS TOOLS

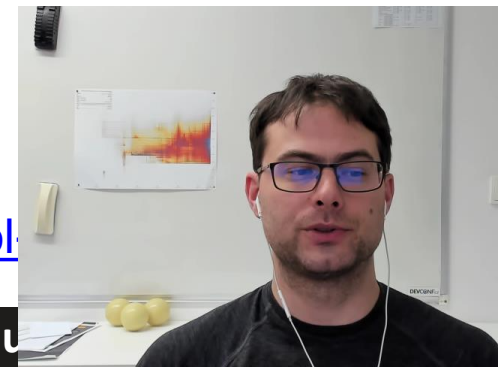


Static analysis tools

- List of static checkers
 - http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html
 - <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 mainly focused on C/C++/Java checkers
 - but tools exists for almost any language

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.lvm.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 <http://www.dwheeler.com/flawfinder/>, Splint <http://www.splint.org/>
 - Rough Auditing Tool for Security (RATS) <http://code.google.com/p/rough-auditing-tool>



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



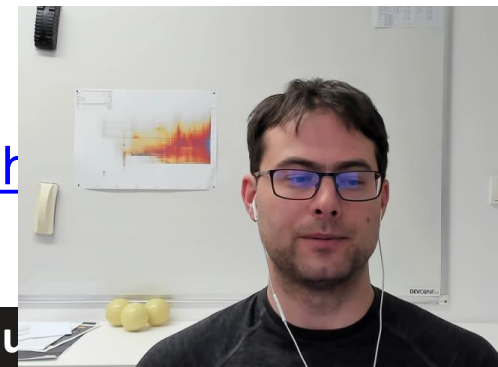
- A tool for static C/C++ code analysis
 - Open-source freeware, <http://cppcheck.sourceforge.net/>
 - Online demo <http://cppcheck.sourceforge.net/demo/>
- Last version 2.3 (2020-12-05)
- Used to find bugs in open-source projects (Linux kernel...)
- Command line & GUI version
- Standalone version, plugin 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?

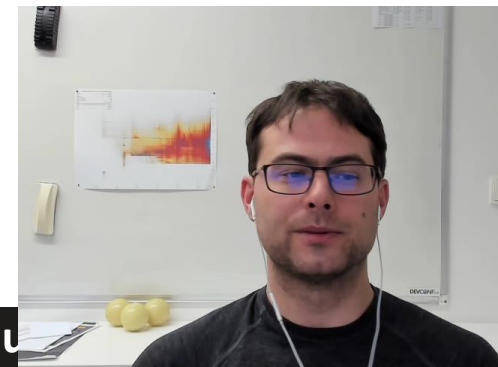
- Bound checking for array overruns
- Suspicious patterns for class
- Exceptions safety
- Memory leaks
- Obsolete functions
- sizeof() related problems
- String format problems...
- See full list

http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page#Ch



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	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 – 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 and print any code, used 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 – simple custom rules (XML)

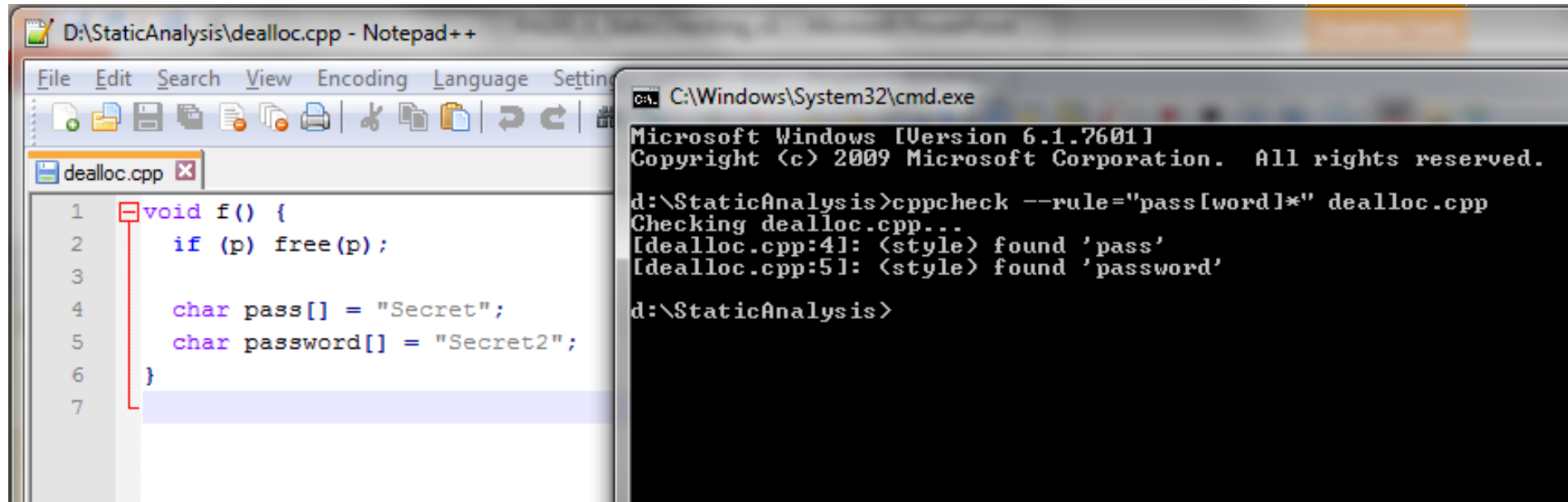
- XML file with regular expression and information
 - pattern to search for
 - information displayed on match

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



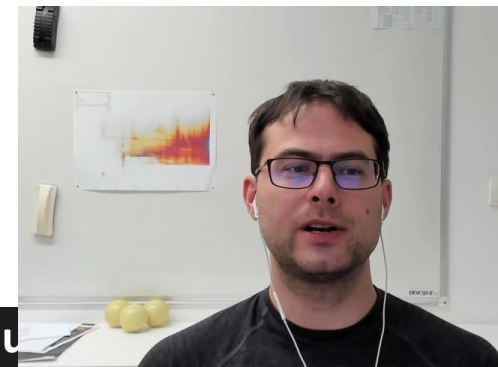
The screenshot shows two windows. The left window is Notepad++ editing a file named 'dealloc.cpp' at 'D:\StaticAnalysis\'. The code in the file is:

```
1 void f() {
2     if (p) free(p);
3
4     char pass[] = "Secret";
5     char password[] = "Secret2";
6 }
7
```

The right window is a Windows command prompt at 'C:\Windows\System32\cmd.exe'. It shows the execution of the command:

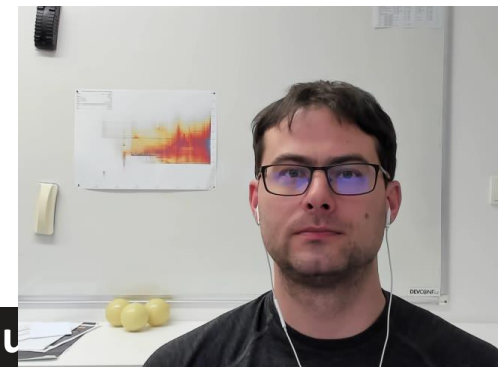
```
d:\StaticAnalysis>cppcheck --rule="pass[word]*" dealloc.cpp
Checking dealloc.cpp...
[dealloc.cpp:4]: (style) found 'pass'
[dealloc.cpp:5]: (style) found 'password'
d:\StaticAnalysis>
```

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



Cppcheck – complex custom rules


- 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: }
```



variable pass has ID @1

```
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
}
```

PREfast - Microsoft static analysis tool

The screenshot shows the Microsoft Visual Studio interface. The main window displays a C++ file named `BufferOverflow.cpp` with the following code:

```

void demoBufferOverflowData() {
    int unused_variable = 3;
    #define NORMAL_USER 'n'
    #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", 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);
}
    
```

The Code Analysis window on the left shows a warning: **C6386 Write overrun**. The description states: "Buffer overrun while writing to 'userName': the writable size is '8' bytes, but '4294967295' bytes might be written." The line explanation indicates that the `userName` array is 8 elements long, but the code attempts to write 4294967295 bytes. The warning is located at line 32 of `bufferoverflow.cpp`.

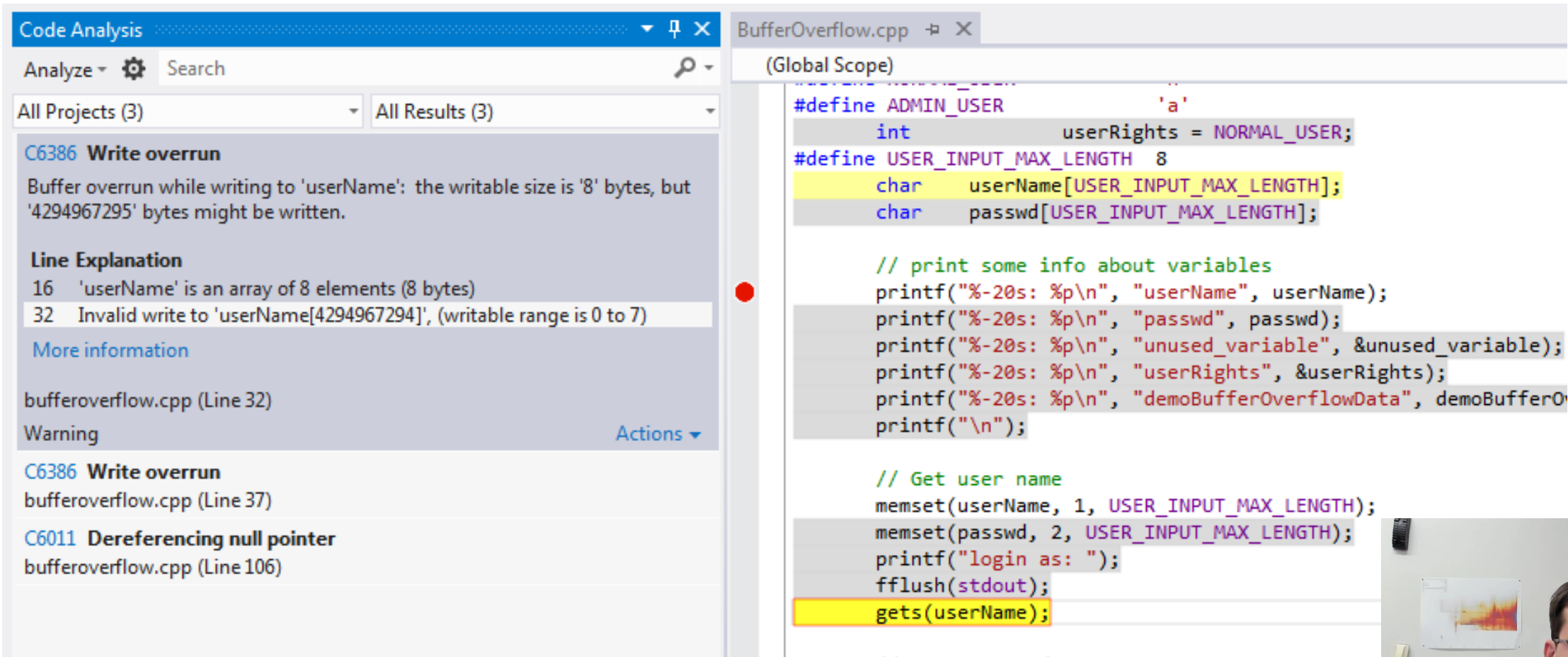
The ANALYZE menu is open, showing options such as "Start Performance Analysis", "Run Code Analysis on Solution", and "Configure Code Analysis for Solution". The "Configure Code Analysis for Solution" option is highlighted with a blue box.



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)

PREfast – example bufferOverflow



The screenshot displays the Visual Studio Code interface. On the left, the 'Code Analysis' window shows a warning for a 'Write overrun' at line 32 of 'bufferoverflow.cpp'. The warning message states: 'Buffer overrun while writing to 'userName': the writable size is '8' bytes, but '4294967295' bytes might be written.' Below this, a 'Line Explanation' section provides details: '16 'userName' is an array of 8 elements (8 bytes)' and '32 Invalid write to 'userName[4294967294]', (writable range is 0 to 7)'. On the right, the 'BufferOverflow.cpp' source code is shown. The code defines 'ADMIN_USER' as 'a', sets 'userRights' to 'NORMAL_USER', and defines 'USER_INPUT_MAX_LENGTH' as 8. It then declares 'userName' and 'passwd' as character arrays of size 'USER_INPUT_MAX_LENGTH'. The code includes several printf statements for debugging, followed by 'memset' calls for 'userName' and 'passwd', and a call to 'gets(userName);' which is highlighted in yellow. A red dot on the left side of the code editor indicates the location of the warning.

```
Code Analysis [Search]
Analyze [Settings] Search [Magnifying Glass]
All Projects (3) All Results (3)
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)

BufferOverflow.cpp [Close]
(Global Scope)
#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", 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 – 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>

The image shows two overlapping windows from Visual Studio. The main window is the 'StaticAnalysis Property Pages' dialog, which is configured for 'Active(Debug)' and 'Active(Win32)'. The 'Code Analysis' section is expanded to 'General', and the 'Enable Code Analysis on Build' checkbox is checked. The 'Rule Set' dropdown is set to 'Microsoft All Rules', and the 'Path' is 'C:\Program Files (x86)\Microsoft Visual Studio 11.0\Team Tool...'. A smaller Notepad window in the foreground shows the XML configuration for the rule set, including the following code:

```
<?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" />
```

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>
- Possibility to write 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 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-JCAIlgTest 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 Page 1 of 1

AlgTestJClient.java

```

265 System.out.println("\n\nSTRONG WARNING: There is possibility tha
266 System.out.println("\n\nWARNING: Your card should be free from o
267 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.

```

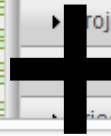
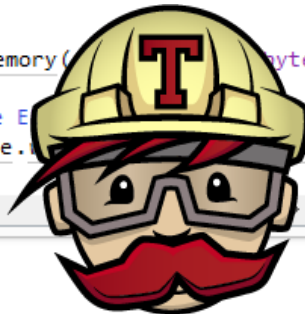
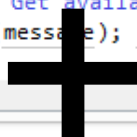
268     answ = Integer.decode(br.readLine());
269 }
270 if (answ == 1) {
271     // Available memory
272     n(0);
273     .TestAvailableEEPROMMemory(byte
274
275     nERROR: Get available E
276     println(message); file.
277

```

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



Code scanning with GitHub + Actions + Codacy

Pull requests Actions Projects Wiki Security Insights Settings

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.

Set up this workflow

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 analysis provided by multiple open source analysis tools.

petsr / pv080_test_cpp

Code Issues Pull requests Actions Projects Wiki **Security** Insights Settings

Overview

Security policy

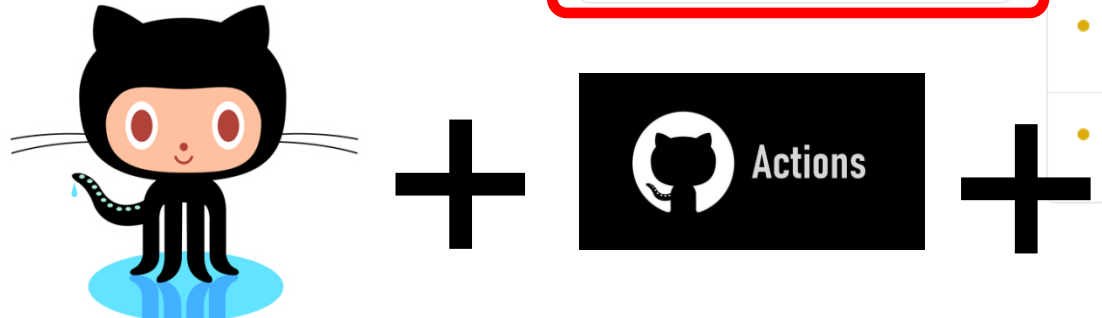
Security advisories 0

Dependabot alerts

Code scanning alerts

Security overview

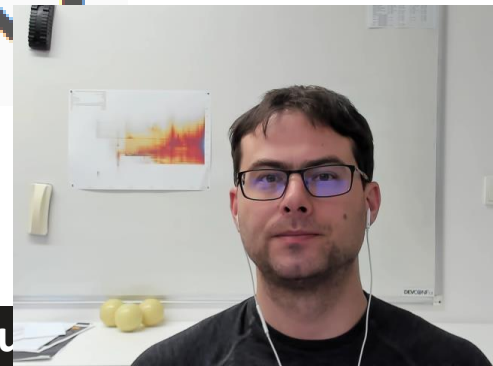
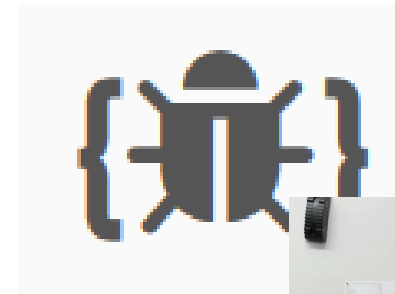
- Security policy**
Define how users should report security vulnerabilities for this repository [Set up a security policy](#)
- Security advisories**
View or disclose security advisories for this repository [View security advisories](#)
- Dependabot alerts**
Get notified when one of your dependencies has a vulnerability
- Code scanning alerts**
Automatically detect common vulnerability and coding errors



SpotBugs



- Static analysis of Java programs (continuation of FindBugs)
- Extended coverage for OWASP Top 10 and CWE
- Current version 4.2.2 (2021-03-03)
 - <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>



FindBugs



- Static analysis of Java programs
- Extended coverage for OWASP Top 10 and CWE
- Current version 3.0.1 (2015-03-06)
 - <http://findbugs.sourceforge.net/>
 - Command-line, GUI, plugins into variety of tools
 - Support for custom rules
- FindSecurityBugs 1.9.0. (2019-03-27)
 - Additional detection rules for FindBugs
 - <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 6.32.0 (2021-02-27)
- 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?

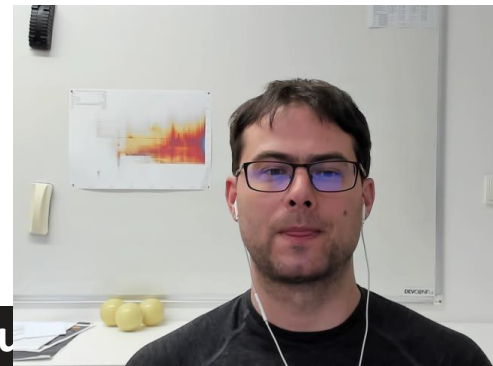
- *“Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.”*



FindBugs™ Fact Sheet



STATIC ANALYSIS IS NOT PANACEA



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'
```

```
printf("Welcome, normal user!\n", userName);
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.
```



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", 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);
}

```

```
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
```

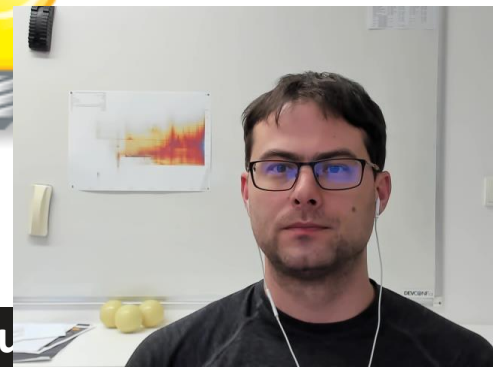
```
bufferoverflow.cpp(13): warn
```

```
Dereferencing NULL pointer 'data_copy'
```

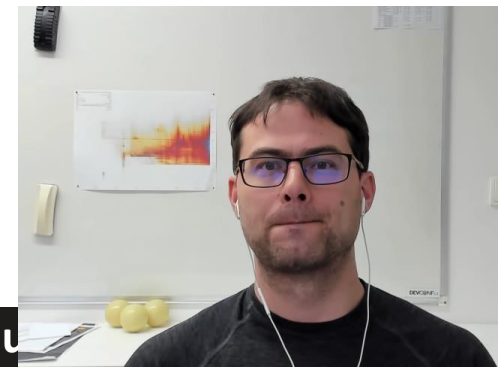


Test suites – vulnerable code, benchmark

- 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/
- Suitable for testing new methods, but NOT for comparison of existing commercial products
 - Public suites, products already optimized for it



SUMMARY



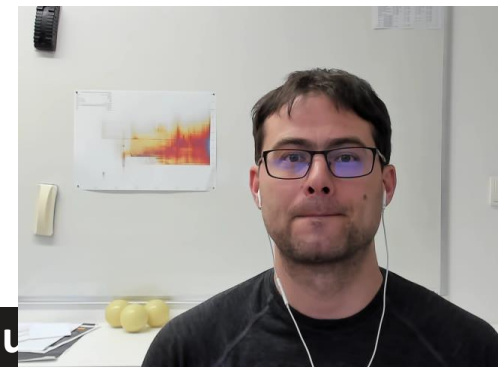
Summary

- 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>
 - <http://go.coverity.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://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.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 2020?
- ...



 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



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

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;
    }
}
```

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;
    }
}
```