

Software Security
Program Analysis with PREfast & SAL

Erik Poll

Digital Security group
Radboud University Nijmegen

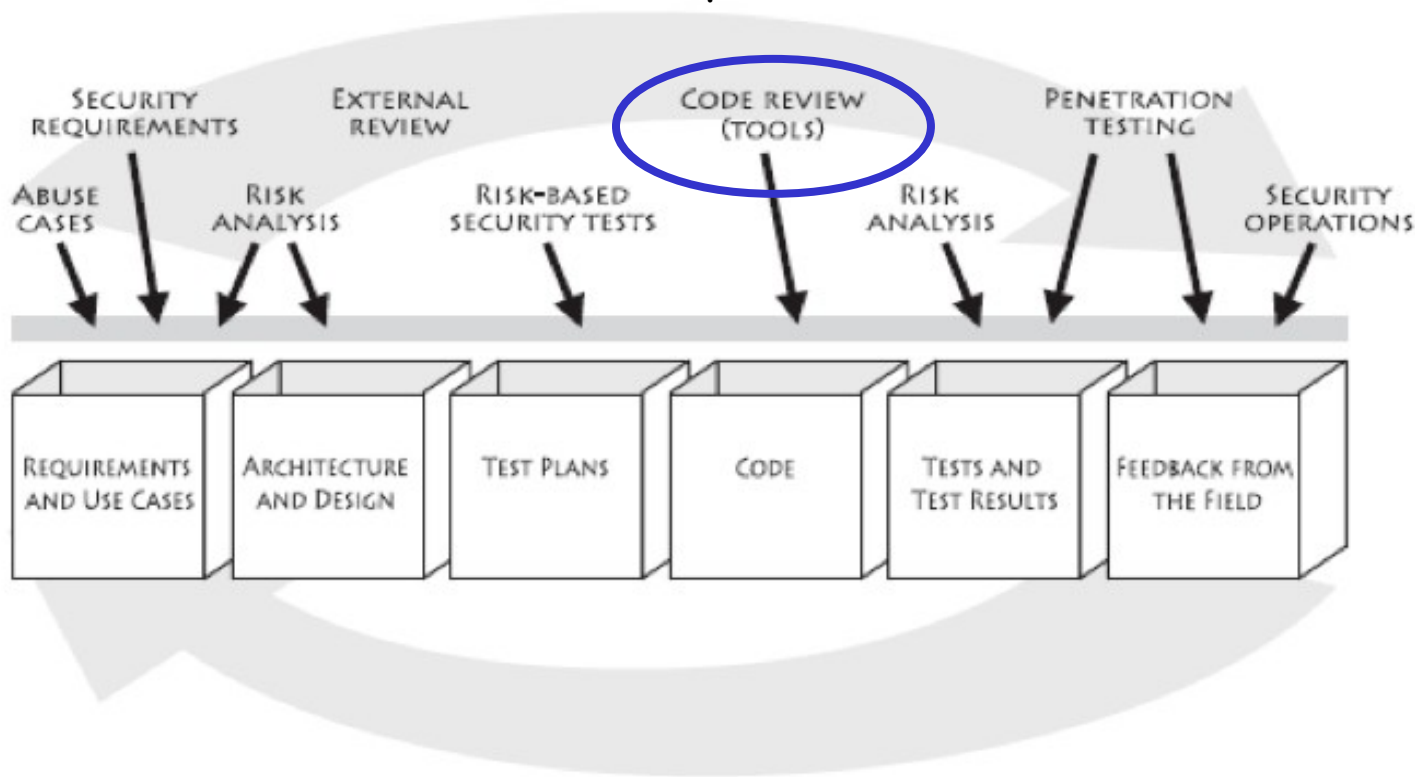
static analysis aka source code analysis aka...

- *Automated analysis at compile time to find potential bugs*
- Broad range of techniques, eg.
 1. simple **syntactic** checks such as `grep` or `CTRL-F`
eg. `grep " gets (" *.cpp`
 2. **type** checking
 3. more advanced analyses take into account the **semantics of programs**
 - using: dataflow analysis, control flow analysis, abstract interpretation, symbolic evaluation, constraint solving, program verification, model checking...

The more lightweight tools are called **source code scanners**

Static analysis/source code analysis in the SDLC

in terms of McGraw's Touchpoints: [code review tools](#)



Why static analysis? (1)

Traditional methods of finding errors:

- testing
- code inspection

Some errors are hard to find by these methods, because they

- arise in unusual circumstances/uncommon execution paths
 - eg. buffer overruns, unvalidated input, exceptions, ...
- involve non-determinism
 - eg. race conditions

Here static analysis can provide major improvement

Quality assurance at Microsoft

- Original process: manual code inspection
 - effective when team & system are small
 - too many paths/interactions to consider as system grew
- Early 1990s: add massive system & unit testing
 - Test took week to run
 - different platforms & configurations
 - huge number of tests
 - Inefficient detection of security holes
- Early 2000s: serious investment in static analysis

False positives & negatives

Important quality measures for a static analysis:

- rate of **false positives**
 - tool complains about non-error
- rate of **false negatives**
 - tool fails to complain about error

Which do you think is worse?

False positives are a killer for usability !!

When is an analysis called

- **sound?** it only finds *real* bugs
- **complete?** it finds *all* bugs

Very simple static analyses

- warning about **bad names and violations of conventions**, eg
 - Java method starting with capital letter
 - C# method name starting with lower case letter
 - constants not written with all capital letters
 - ...
- enforcing other (company-specific) naming conventions and coding guidelines
 - this is also called **style checking**

More interesting static analyses

- warning about **unused variables**
- warning about **dead/unreachable code**
- warning about **missing initialisation**
 - possibly as part of language definition (eg Java) and checked by compiler
 - this may involves
 - **control flow analysis**
 - if (b) { c = 5; } else { c = 6; } initialises c
 - if (b) { c = 5; } else { d = 6; } does not
 - **data flow analysis**
 - d = 5; c = d; initialises c
 - c = d; d = 5; does not

Example source code analysis tools

- for Java: *CheckStyle*, *PMD*, *Findbugs*,....
- for C(++) from Microsoft: *PREfix*, *PREfast*, *FxCop*
- somewhat outdated, but free tools focusing on security
 - *ITS4* and *Flawfinder* (C,C++), *RATS* (also Perl,PHP)
- commercial
 - *Coverity* (C,C++), *Klocwork* (also Java), *PolySpace* (also Ada)
- for web-applications
 - commercial: *Fortify* (PHP, Java, .Net)
 - open source: *Code Crawler*, *Orizon*, *Pixy*

good fun
to download
and try out!

Such tools can be useful, but... a fool with a tool is still a fool

PREfast & SAL

PREfast & SAL

- Developed by Microsoft as part of major push to improve quality assurance
- **PREfast** is a lightweight static analysis tool for C(++)
 - only finds bugs within a single procedure
- **SAL** (Standard Annotation Language) is a language for annotating C(++) code and libraries
 - SAL annotations improve the results of PREfast
 - more checks
 - more precise checks
- PREfast & SAL of particular interest to device driver writers

PREfast checks

- library function usage
 - depreciated functions
 - eg gets()
 - correct use of functions
 - eg does format string match parameter types?
- coding errors
 - eg using = instead of == in an if-statement
- memory errors
 - assuming that malloc returns non-zero
 - going out of array bounds

PREfast example

```
__Check_return__ void *malloc(size_t s);
```

__Check_return__ means that caller *must* check the return value of `malloc`

SAL annotations for buffer parameters

- `_In_` The function reads from the buffer. The caller provides the buffer and initializes it.
- `_Inout_` The function both reads from and writes to buffer. The caller provides the buffer and initializes it.
- `_Out_` The function will only write to the buffer. The caller must provide the buffer, and the function will initialize it..

The tool can then check if (unitialised) output variables are not read before they are written

SAL annotations for buffer sizes

specified with suffix of `_In_` `_Out_` `_Inout_` `_Ret_`

- `bytecount_(size)` or `bytecap_(size)`
buffer size in bytes
- `count_(size)` or `cap_(size)`
buffer size in elements
- extra suffix `_c_` if size is a constant

`count`, `bytecount` used for inputs, ie. `_In_`

`cap`, `bytecap` used for output/results, ie. `_Out_` , `_Ret_`

SAL annotations for nullness of parameters

specified with prefix

- `opt_`
parameter may be null, and procedure will check for this
- no prefix means pointer may not be null

Note that this is moving towards **non-null by default**

SAL annotations for buffer sizes

Warning: SAL syntax was changed in 2009. Eg

<code>_In_bytecount_(..)</code>	used to be	<code>__in_bcount(..)</code>
<code>_Out_count_(..)</code>	used to be	<code>__out_ecount(..)</code>
<code>_Ret_count_(..)</code>	used to be	<code>__ecount(..)</code>

Some of the documentation you may find online still uses old syntax.

PREfast example

```
void * memset(  
_Out_bytecount(len) char *p,  
int v,  
size_t len);
```

_Out_bytecount(len) specifies that

- `memset` will only write the memory at `p`
- it will write `len` bytes

Example annotation & analysis

```
void work() {  
    int elements[200];  
    wrap(elements, 200);  
}  
int *wrap(int *buf, int len) {  
    int *buf2 = buf;  
    int len2 = len;  
    zero(buf2, len2);  
    return buf;  
}  
void zero( int *buf,  
          int len){  
    int i;  
    for(i = 0; i <= len; i++) buf[i] =  
        0;  
}
```

Example annotation & analysis

Building and solving constraints

```
void work() {  
  int elements[200];  
  wrap(elements, 200);  
}
```

```
_Ret_cap_(len) *wrap(  
  _Out_cap_(len) int *buf,  
  int len) {
```

```
  int *buf2 = buf;  
  int len2 = len;  
  zero(buf2, len2);  
  return buf;  
}
```

```
void zero(_Out_cap_(len) int *buf,
```

```
  int len) {
```

```
  int i;  
  for(i = 0; i <= len; i++) buf[i] = 0;  
}
```

1. Builds **constraint**
len = length(buf)

2. Checks **contract** for
call to zero

3. Checks **contract** for return

4. Builds **constraints**
len = length(buf)
i ≤ len

5. Checks

0 ≤ i < length(buf)



SAL pre- and postconditions

```
#include </prefast/SourceAnnotations.h>
[SA_Post( MustCheck=SA_Yes )] double* CalcSquareRoot
(
    [SA_Pre( Null=SA_No )] double* source,
    unsigned int size
)
```

- `[SA_Post (MustCheck=SA_Yes)]` requires caller to check the return value of `CalcSquareRoot`
 - alternative syntax for `_Check_return_`
- `[SA_Pre (Null=SA_No)]` requires caller to pass non-null parameter `source`

Tainting annotations in pre/postconditions

- `[SA_Pre(Tainted=SA_Yes)]`

This argument is tainted and cannot be trusted without validation.

- `[SA_Pre(Tainted=SA_No)]`

This argument is not tainted and can be trusted

- `[SA_Post(Tainted=SA_No)]`

As above, but as postcondition

Benefits of annotations

- Annotations express design intent
 - for human reader & for tools
- Adding annotations you can find more errors
- Annotations improve precision
 - ie reduce number of false negatives and false positives
 - because tool does not have to guess design intent
- Annotations improve scalability
 - annotations isolate functions so they can be analysed one at a time
 - allows intra-procedural (local) analysis instead of inter-procedural (global) analysis

Drawback of annotations

- The effort of having to write them...
 - who's going to annotate the millions of lines of (existing) code?
- Practical issue of motivating programmers to do this
- Microsoft approach
 - requiring annotation on checking in new code
 - rejecting any code that has `char*` without `_count()`
 - making annotations natural
 - incremental approach, in two ways:
 - beginning with few core annotations
 - checking them at every compile, not adding them in the end
 - build tools to infer annotations, eg `SALinfer`
 - unfortunately, not available outside Microsoft

Static analysis in the workplace

- Static analysis is not for free
 - commercial tools cost money
 - all tools cost time & effort to learn & use

Criteria for success

- acceptable level of false positives
 - acceptable level of false negatives also interesting, but less important
- not too many warnings
 - this turns off potential users
- good error reporting
 - context & trace of error
- bugs should be easy to fix
- you should be able to teach tool
 - to suppress false positives
 - add design intent via assertions

(Current?) limitations of static analysis

- The heap poses a major challenge for static analysis
 - heap is a very dynamic structure evolving at runtime: what is a good abstraction at compile-time?
- Many static analysis will disregard the heap completely
 - note that all the examples in these slides did
 - this is then a source of false positives and/or false negatives