Software Security

Static Analysis aka Source code analysis

Erik Poll

Digital Security group

Radboud University Nijmegen

static analysis aka source code analysis

Automated analysis *at compile time* to find potential *bugs*

Broad range of techniques, from light- to heavyweight:

1. simple syntactic checks such as **`grep`** or **`CTRL-F`**

   eg. **`grep " gets(" *.cpp`**

2. type checking

3. more advanced analyses take into account program semantics

   - 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

Demos

- Findbugs

  open source static checker for Java

- PREfast

  Microsoft tool to analyse C(++)

  PREfast is included in *some* versions of VisualStudio

Why static analysis?

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, ie no false pos
- complete? it finds *all* bugs, ie no false neg

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

A nice Java tool for style checking to try is CheckStyle

More interesting static analyses

- warning about unused variables
- warning about dead/unreachable code
- warning about missing initialisation
  - possibly as part of language definition (as it is for Java) and checked by compiler
  - this may involve
    - 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

Limits of static analyses

Does

```
    if (i < 5 ) { c = 5; }
    if (i < 0) || (i*i > 20){ c = 6; }
```

initialise c?

Many analyses become hard – or *undecidable* - at some stage

Analysis tools can then...

- report that they “DON’T KNOW”
- give a (possibly) false positive
- give a (possibly) false negative

The PREfast tool can do some arithmetic

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 (C(++), Java), PolySpace (C(++) , Ada)
- for web-applications

  commercial: Fortify, Microsoft CAT.NET, CheckMarx...

  open source: RIPS, OWASP Orizon, ...

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

FindBugs

FindBugs

Source code analyser for Java

- very simple to use
- only requires the compiled byte code (ie. class or jar files)

It distinguishes different types of bugs

- bad practice, correctness, multi-threading, performance, security, ..

and different priorities

- high, medium, or low

demo

PREfast & SAL

Remember

- buffer overflows are a major source of security problems
- buffer overflow vulnerabilities can be really hard to spot

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 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

demo

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

Possible (non)nullness is specified with prefix

- `opt_`

  parameter may be null, and procedure will check for this

- no prefix means pointer is not (meant to be) null

Note that this is moving towards non-null by default

Need for annotations

- PREfast checks one procedure at the time:
  - aka intra-procedural and not interprocedural
- Advantage: this is more efficient and scales better
  - An interprocedural analyses has to do whole program analysis
- Disavantage: analysis per procedure cannot see problems across procedures boundaries
  - Partial solution: introducing annotations to specify properties at procedure boundaries

SAL annotations for buffer sizes

Warning: SAL syntax was changed in 2009. Eg

`_In_bytecount_(..)` used to be `__in_bcount(..)`

`_Out_count_(..)` used to be `__out_ecount(..)`

`_Ret_count_(..)` used to be `__ecount(..)`

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

How does it work?

```
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;
}
```

How does it work?

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

Tool will build and solve constraints

1. Builds constraint
   len = length(buf)
2. Checks contract for call to zero
3. Checks contract for return

1. Builds constraints
   len = length(buf)
   i ≤ len
2. Checks
   0<=i < length(buf)

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()
  - 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

Tainting analysis

Some source code analysis tools do an analysis called tainting:

1. User input is marked as tainted
2. Tainting is propagated, using data flow analysis
3. If tainted data ends up as argument in dangerous API calls, the tools complains.

Examples of dangerous API calls are

- calling the operation system, eg. with **`system(...)`** in C
- calling the SQL database

Static analysis in the workplace

Static analysis is not for free

- commercial tools cost money
- all tools cost time & effort to learn to 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 the tool
  - to suppress false positives
  - add design intent via assertions

(Current?) limitations of static analysis

- *The heap* poses a major challenge for static analysis
  - the 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

In some coding standards for safety- or security-critical code, eg MISRA-C, it is not allowed to use the heap aka dynamic memory at all

Try them!

- Try out some of these tools on some code you wrote!
- Nice ones to try
  - for Java: Findbugs, CheckStyle, and PMD
  - for web applications: RIPS & CheckMarx
  - for C and C++: RATS, ITS4, FlawFinder, PREfast

- PREfast is included in *some* versions of Visual Studio. You have to give the command line option **/analyze** to switch it on. See http://www.cs.ru.nl/~erikpoll/ss/project1/running_prefast.html