Software Security

Program Analysis with PREfast & SAL

Erik Poll

Digital Security group

Radboud University Nijmegen

Recap from last week

- Buffer overflows notorious source of security flaws in C(++) code
  - Classic example: attacker *overflows buffer* on the stack, to inject his own machine code (aka shell code) and corrupt control data (ie. the return address) to execute this code

    Preventable by distinguishing W ⊕ X : (non)executable memory
  - Or: attacker *corrupts control data* to execute other code (library calls, or parts of library calls)
  - Or: attacker *corrupts some other data* on stack or heap, or *reads confidential data* that should be confidential (using buffer overflow, double free, ...)
- Spotting such flaws in code is hard

static analysis aka source code analysis aka...

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

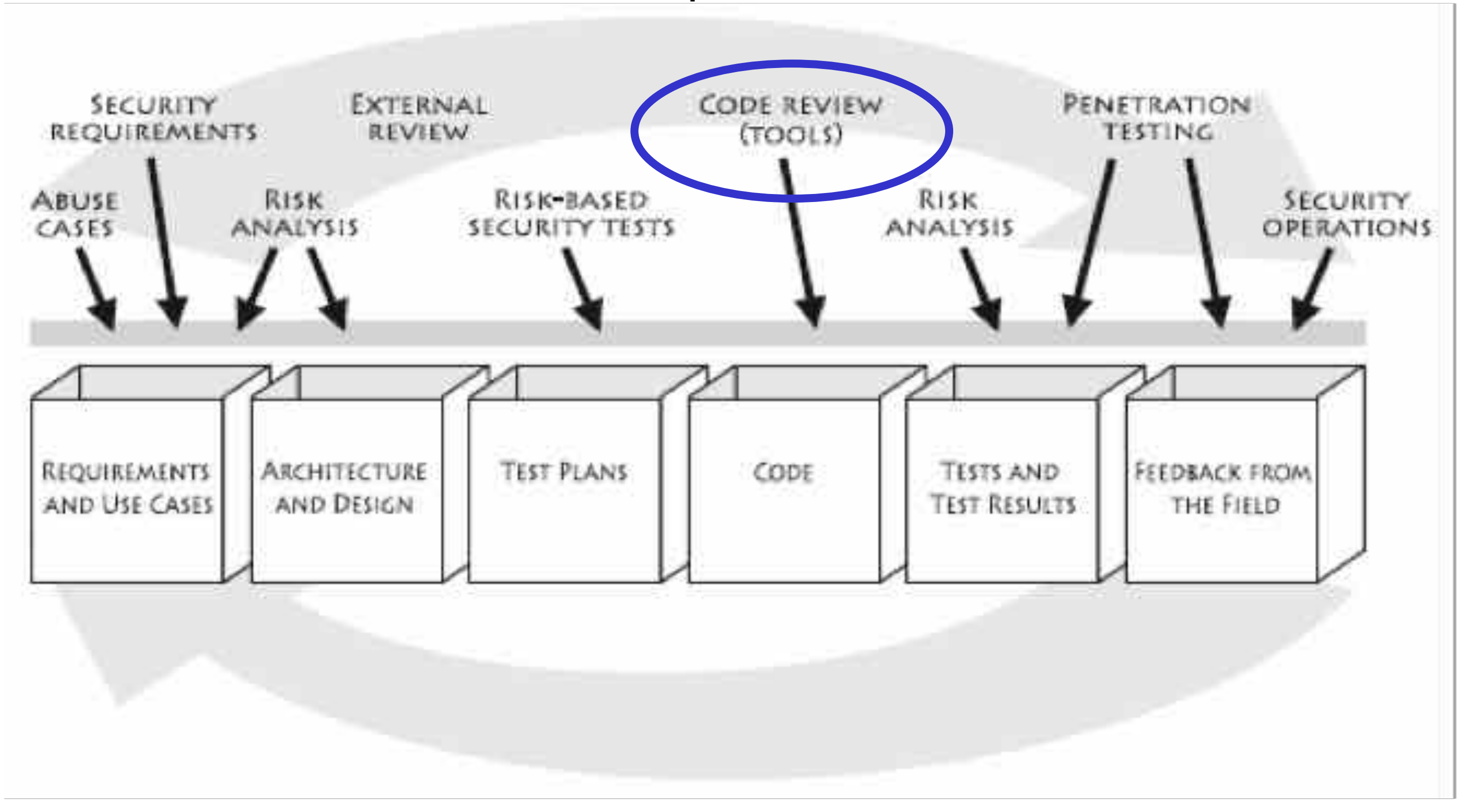
3. more advanced analyses take into account program semantics

   - 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

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 ! !

Alternative terminology. When is an analysis called

- sound? it only finds *real* bugs, ie no false positives
- complete? it finds *all* bugs, ie. no false negatives

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 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
```

Spot the defect!

```
BOOL AddTail(LPVOID p) {
 ...
  if(queue.GetSize() >= this->_limit);
  {
   while(queue.GetSize() > this->_limit-1)
   {
     ::WaitForSingleObject(handles[SemaphoreIndex],1);
     queue.Delete(0);
   }
}
```

Suspicious code in xpdfwin found by PVS-Studio (*www.viva64.com)*.

```
V529 Odd semicolon ';' after 'if' operator.
```

Note that this is a very simple syntactic check!

Spot the security flaw!

```
static OSStatus SSLVerifySignedServerKeyExchange (SSLContext
*ctx, bool isRsa, SSLBuffer signedParams,uint8_t *signature,
UInt16 signatureLen)
{ OSStatus  err;
  ..
 if((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
 if((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
 if((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
  ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
}
```

iOS goto flaw in SSL

How to prevent such a bug?

- enforce coding style to always write { },
  even around one-statement code blocks

```
 if ( ... ) {
        goto fail; }
        goto fail;
```

- use a static analysis to warn about dead code

Spot the defects!

```
void start_engine_control() {
 char*  buf2 = malloc (2*SOME_CONSTANT);
 char*  buf = malloc (SOME_CONSTANT);
 start_engine();
 memset(buf2, 0, SOME_CONSTANT);
      // initialise first half of buf2 to 0

 // main loop
 while (true) do {
   get_readings(buf,buf2);
   perform_engine_control(buf,buf2);
 }
}
```

Spot the defects!

```
void start_engine_control() {
 char*  buf2 = malloc (2*SOME_CONSTANT);
 char*  buf = malloc (SOME_CONSTANT);
 start_engine();
 memset(buf2, 0, SOME_CONSTANT);
      // initialise first half of buf2 to 0

 // main loop
 while (true) do {
   get_readings(buf,buf2);
   perform_engine_control(buf,buf2);
 }
}
```

possible integer overflow
(hard to check for code analyser, but for a constant is may be doable)

No check if malloc succeeded!!
(easier to check syntacially)

Check you malloc's!

```
void start_engine_control() {
 ...
 char*  buf = malloc (SOME_CONSTANT);
 if (buf == NULL) { //now what?!?!?
         exit(0); // or something more graceful?
 }
 ...
 start_engine();
 perform_engine_control(buf);
```

Typically, the place where the malloc fails is the place to think about what to do.

We could not check the result of malloc here and simply let **`perform_engine_control`** segfault, or let it check for null arguments, but there we have even less clue on what to do..

Limits of static analyses

Does

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

initialise c?

Many analyses become hard - if not 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(++): PVS-Studio
- 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,...

  open source: RIPS, OWASP Orizon, ...

easy & 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
  - deprecated 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`

PREfast motivation

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

void memcpy( char *dest,
             char *src,
             size_t count);
```

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_`**

- **`cap_(size)`** the *writeable* size in elements
- **`bytecap_(size)`** the *writeable* size in bytes

- **`count_(size)`** **`bytecount_(size)`**

  the *readable* size in elements

`count` and **`bytecount`** should be only be used for inputs, ie. parameter declared as **`_In_`**

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 may not be null

Note that this is moving towards non-null by default

PREfast example

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

`_Out_cap_(len)` specifies that

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

PREfast example

```
void memcpy( char _Out_cap_(count) *dest,
             char _In_count_(count) *src,
             size_t count);
```

So **`memcopy`** will read **`src`** the and write to **`dest`**

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

```
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
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)
```

Here `[SA_Post (MustCheck=SA_Yes)]`

requires caller to check the return value of CalcSquareRoot

(this is an alternative syntax for `_Check_return_`)

and `[SA_Pre (Null=SA_No)]`

requires caller to pass non-null parameter **source**

Tainting annotations in pre/postconditions

You can specify pre- and postconditions to express if inputs or outputs of a methods maybe tainted

Here tainted means this is untrusted user input, which may be malicious

SAL specifications for tainting:

- `[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 for the result

Warning: changing SAL syntax

- SAL syntax keeps changing – the current version is 2.0

  For the individual exercise, stick to the syntax described in these slides & on the webpage for the exercise.

- PREfast behaviour can be a bit surprising when you use count instead of cap or when you use bytecap instead of cap

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

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

Note that 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