coverity®

# Static Analysis

More than finding bugs

Bob Archer

coverity®

# What do we do?

# Look for defects in code

(in C/C++, Java and C#)

# A simple example – find the bug

```
void foo0( int* p, int i, int j )
{
    if( i > 7 )
        p = 0;

    *p = j;
}
```

# Slightly more subtle – find the bug

```
void foo( int* );

void foo5( int* p, int i, int j )
{
    if( p )
        foo( p );

    *p = j;
}
```

# Checker examples

- FORWARD_NULL
- DEADCODE
- COPY_PASTE_ERROR
- LOCK_INVERSION
- RESOURCE_LEAK
- NESTING_INDENT_MISMATCH

# DEADCODE

```
void deadcode( int i )
{
    if( i != 10 || i != 12 ) {
        return;
    }

    ++i;
}
```

# DEADCODE

```
void deadcode( int i )
{
    if( i != 10 || i != 12 ) {
        return;
    }

    ++i;
}
```

The condition is always true

# COPY_PASTE_ERROR

```
struct S {
    int x, y;
};

void copyPasteError( bool b, int z ) {
    S s1;
    s1.x = 0;
    s1.y = 0;

    if( b )
        s1.x = s1.x + z;
    if( b )
        s1.y = s1.x + z;
}
```

# COPY_PASTE_ERROR

```
struct S {
    int x, y;
};

void copyPasteError( bool b, int z ) {
    S s1;
    s1.x = 0;
    s1.y = 0;

    if( b )
        s1.x = s1.x + z;
    if( b )
        s1.y = s1.x + z;
}
```

# LOCK_INVERSION

```
int a, b;

void test1_ok() {
    _spin_lock(&a);
    _spin_lock(&b);
}
void test1_good() {
    _spin_lock(&a);
    _spin_lock(&b);
}
void test1_bad() {
    _spin_lock(&b);
    _spin_lock(&a);
}
```

# LOCK_INVERSION

```
int a, b;

void test1_ok() {
    _spin_lock(&a);
    _spin_lock(&b);
}
void test1_good() {
    _spin_lock(&a);
    _spin_lock(&b);
}
void test1_bad() {
    _spin_lock(&b);
    _spin_lock(&a);
}
```

# RESOURCE_LEAK

```
void test2(bool b)
{
    int* p;

    p = malloc(10);
    if (b)
        free(p);
}
```

# NESTING_INDENT_MISMATCH

```
void nesting_indent_mismatch(int x)
{
    if (x == 0)
        x = foo();
        bar(x);
}
```

# NESTING_INDENT_MISMATCH

```
void nesting_indent_mismatch(int x)
{
    if (x == 0)
        x = foo();
        bar(x);
}
```

# Checkers

ARRAY_VS_SINGLETON
ASSERT_SIDE_EFFECT
ATOMICITY
BAD_ALLOC_ARITHMETIC
BAD_ALLOC_STRLEN
BAD_COMPARE
BAD_EQ
BAD_EQ_TYPES
BAD_FREE
BAD_OVERRIDE
BAD_SHIFT
BAD_SIZEOF
BUFFER_SIZE
CALL_SUPER
CHAR_IO
CHECKED_RETURN
CHROOT
COM.BAD_FREE
COM.BSTR.CONV
COPY_PASTE_ERROR
COPY_WITHOUT_ASSIGN
CTOR_DTOR_LEAK
DEADCODE
DELETE_ARRAY
DELETE_VOID
ENUM_AS_BOOLEAN
EVALUATION_ORDER
FORWARD_NULL
GUARDED_BY_VIOLATION
INFINITE_LOOP
INVALIDATE_ITERATOR
LOCK
LOCK_INVERSION
MISRA_CAST
MISSING_BREAK
MISSING_LOCK
MISSING_RETURN
NEGATIVE_RETURNS
NO_EFFECT
NULL_RETURNS
OPEN_ARGS
ORDER_REVERSAL
OVERRUN
PARSE_ERROR
PASS_BY_VALUE
READLINK
RESOURCE_LEAK
RETURN_LOCAL
REVERSE_INULL
REVERSE_NEGATIVE
SECURE_CODING
SECURE_TEMP
SELF_ASSIGN
SIGN_EXTENSION
SIZEOF_MISMATCH
SLEEP
STACK_USE
STRAY_SEMICOLON
STREAM_FORMAT_STATE
STRING_NULL
STRING_OVERFLOW
STRING_SIZE
SWAPPED_ARGUMENTS
TAINTED_SCALAR
TAINTED_STRING
TOCTOU
UNCAUGHT_EXCEPT
UNINIT
UNINIT_CTOR
UNREACHABLE
UNUSED_VALUE
USE_AFTER_FREE
USER_POINTER
VARARGS
VOLATILE_ATOMICITY
WRAPPER_ESCAPE

# A talk in three parts

Technical

(how do we find defects?)

Philosophical

(what is a defect anyway?)

Psychological

(how do we persuade a programmer that there really is a defect?)

# Technical

## How do we find defects?

# What is static analysis?

# What is static analysis?

Looking for defects *without* running the code

(we analyze the code itself, not the execution of that code)

# What is static analysis?

Analysis of *all* paths (in theory)

(not just those encountered during execution of a test suite)

# Problems

Complete analysis ≡ Halting problem

# Problems

Complete analysis ≡ Halting problem

Huge state space

# Problems

Complete analysis ≡ Halting problem

Huge state space

Combinatorial explosion

# Solution

Abstract Interpretation

# Solution

Abstract Interpretation

(simplify, simplify, simplify)

# New problems

What do we simplify?

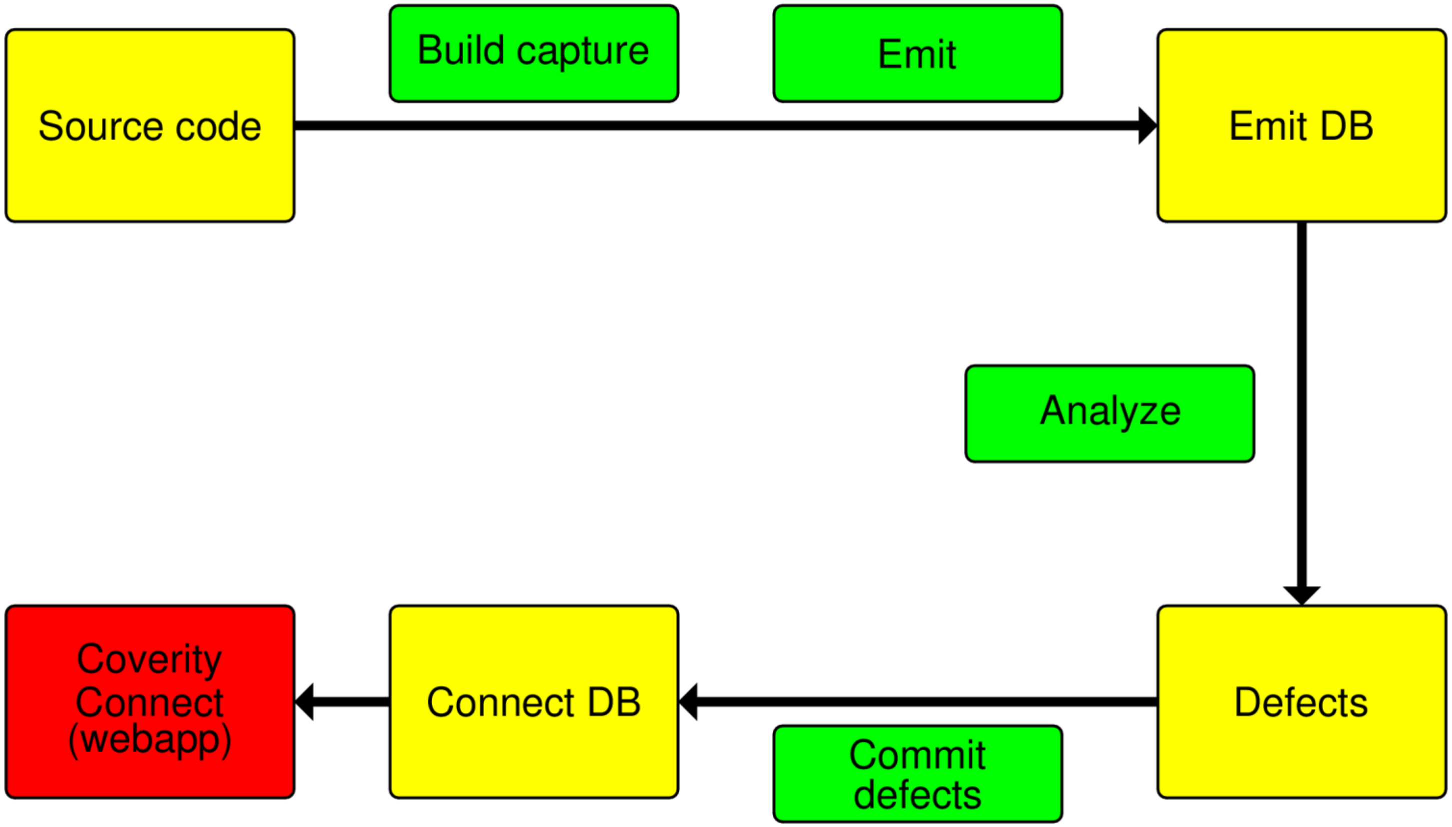
How do we simplify it?

How much do we simplify it?

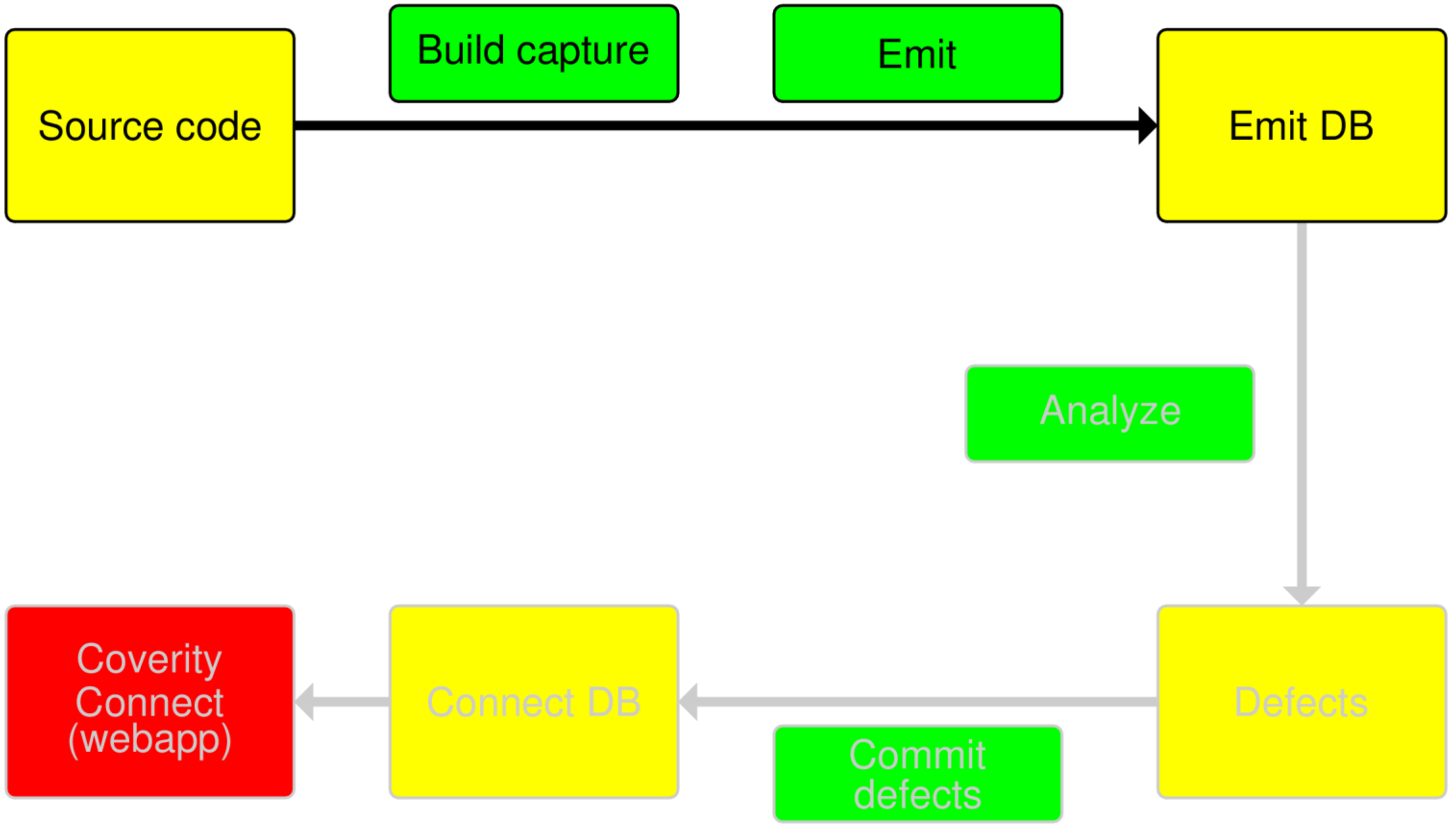
# The bane of our existence

| | Coverity does not report bug | Coverity reports bug |
|---|---|---|
| **Bug exists in code** | False Negative | True Positive |
| **Bug does not exist in code** | True Negative | False Positive |

# Overall picture



# Build capture & emit



# Build capture

- Wrap the user's build
  - Files
  - Order
  - Command line options
  - Compiler used

- Just building is a hard problem

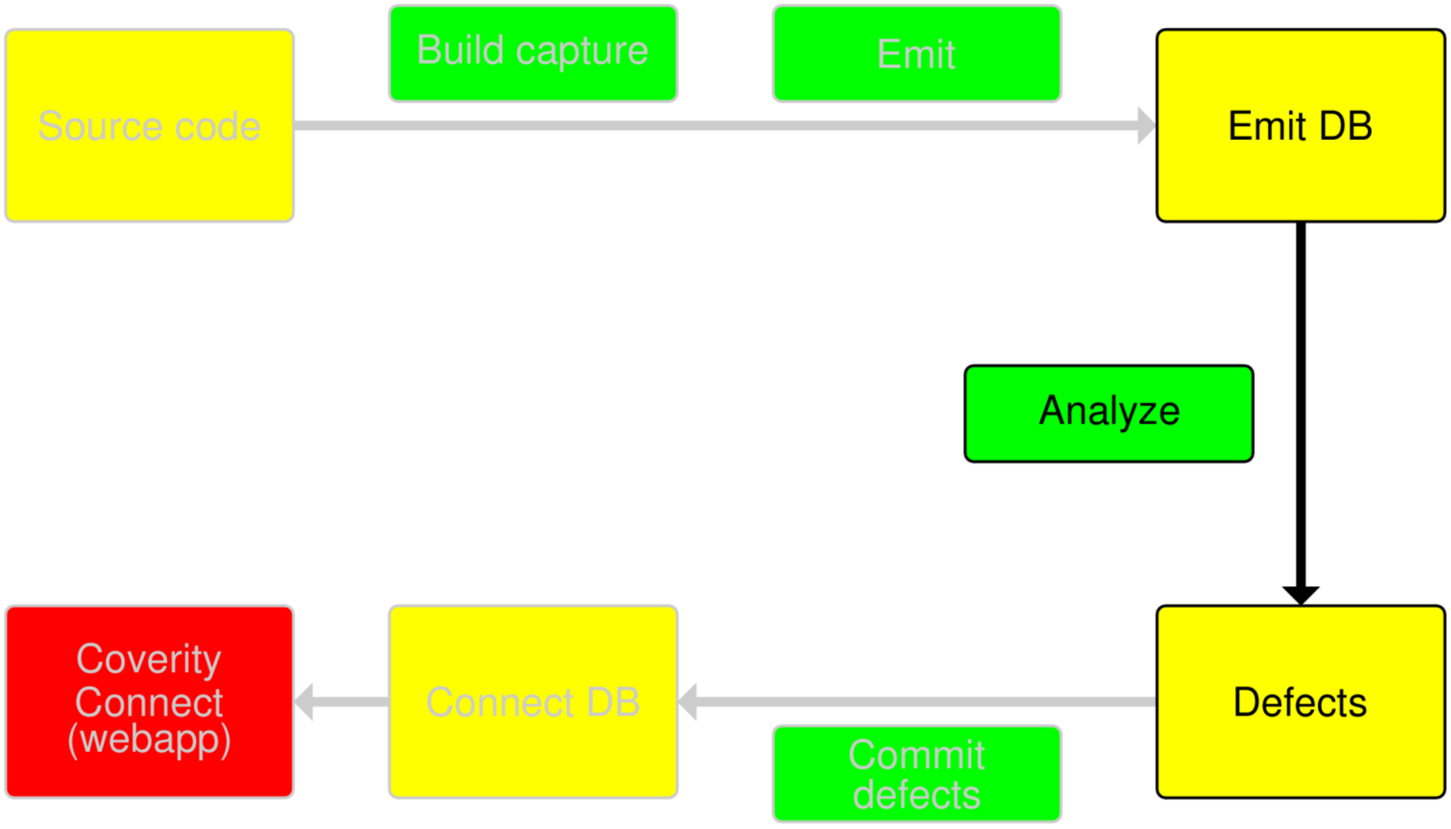
# Emit

- Our own compiler front end
- Produces an AST + other data we want

# Other data?

- Indentation information
- Unnecessary casts that are (usually) thrown away by Mono front end
- The complete list of every file that was compiled along with all of the options used so that we don’t have to run the build system again.

# Analysis



coverity®

# Analysis

Look for known defect patterns in the AST

Report them

# Intraprocedural / interprocedural

- Intraprocedural
  - No (or limited) knowledge of other functions
  - Poor for data sensitive checkers
  - Usually simple & fast

- Interprocedural
  - Knows a lot about other functions
  - Good for tracking data flow between functions
  - Abstract interpretation – create simplified models of other functions
  - Complex, slow, necessary

# Flow insensitive / flow sensitive

- Flow insensitive
  - Things that are always true
  - Mostly intraprocedural
  - High certainty – makes few assumptions

- Flow sensitive
  - Things that are conditionally true
  - Tracks the evolution of (simplified) state
  - Intraprocedural or interprocedural
  - Less certainty – relies on abstractions

# Flow sensitive analysis

- Loops are a problem
- Continue until nothing more to be learned
- Trade off between speed and analysis fidelity

# False path pruning

```
void f(int x) {
    int y;
    if (x)
        y = 1; /* A */
    else
        y = 2; /* B */
    ...        /* C */
    if (x)
        ++y;    /* D */
    else
        --y;    /* E */
}
```

# False path pruning

```
void f(int x) {
    int y;
    if (x)
        y = 1; /* A */
    else
        y = 2; /* B */
    ...        /* C */
    if (x)
        ++y;   /* D */
    else
        --y;   /* E */
}
```

Feasible:

A – C – D

B – C – E

Infeasible

A – C – E

B – C – D

# Is there a defect here?

```
char foo( int fd )
{
    char buf;
    lseek( fd, 0, SEEK_SET );
    read( fd, &buf, /*nbyte*/ 1 );
    return buf;
}
```

# Is there a defect here?

```
char foo( int fd )
{
    char buf;
    lseek( fd, 0, SEEK_SET );
    read( fd, &buf, /*nbyte*/ 1 );
    return buf;
}
```

# Is there a defect here?

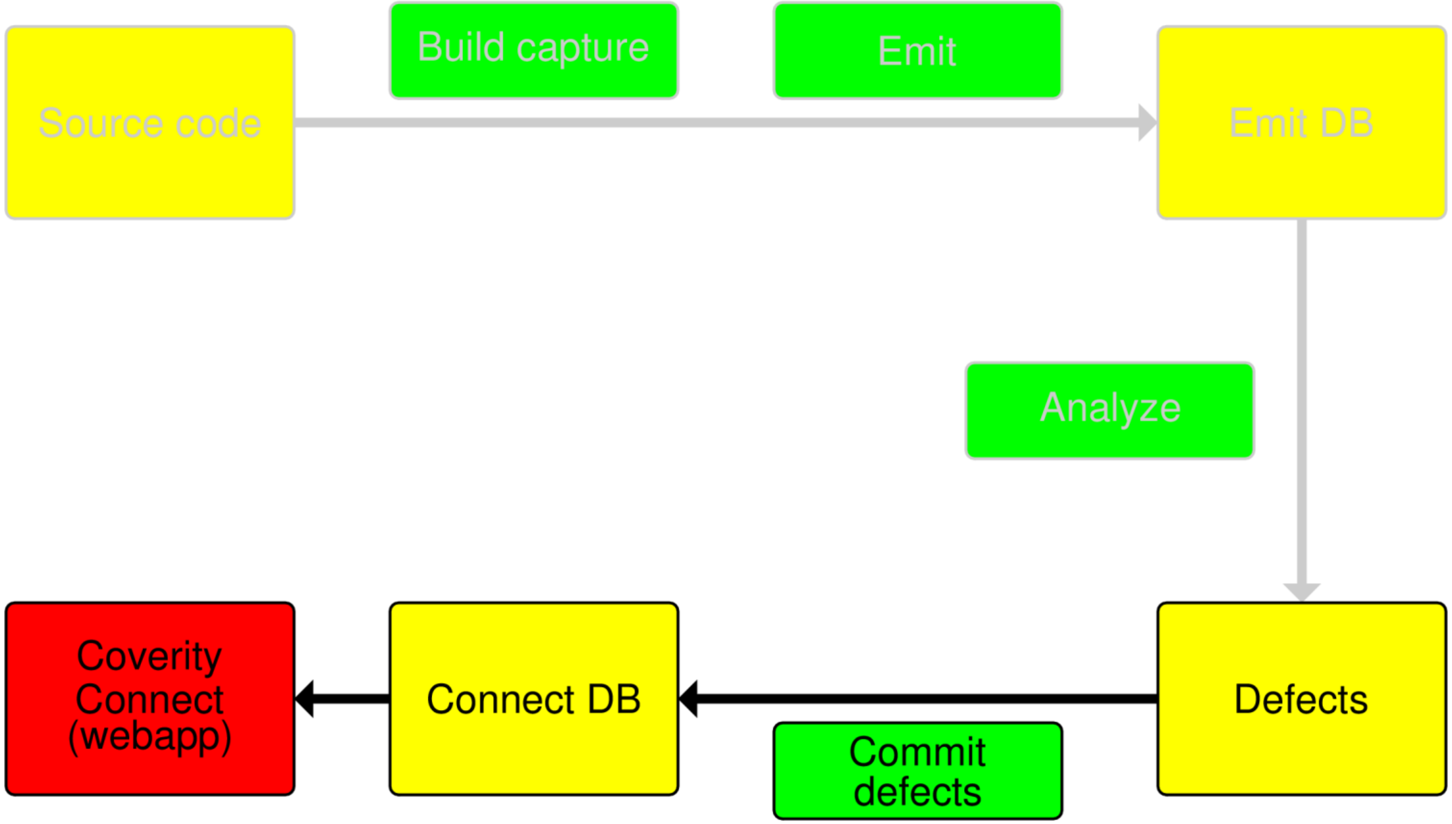
```
char foo( int fd )
{
    char buf;
    lseek( fd, 0, SEEK_SET );
    read( fd, &buf, /*nbyte*/ 1 );
    return buf;
}
```

lseek is called 10 times in the code. Its return value is checked in the other 9 places.

# Statistical checkers

- Look for consistency
- Return value of lseek checked 9 times out of 10 – probable error

# Commit


Source code
Build capture
Emit
Emit DB
Analyze
Defects
Commit defects
Connect DB
Coverity Connect (webapp)

# Commit

- Adds defects to the database

# Commit

- Adds defects to the database
- Remember that we’re not just doing this once.
  - The code might undergo many revisions and have analysis re-run each time
  - Once a bug has been committed we don’t want it reported as a separate entry. (Lose categorization information)

# Commit

- Adds defects to the database
- Remember that we're not just doing this once.
  - The code might undergo many revisions and have analysis re-run each time
  - Once a bug has been committed we don't want it reported as a separate entry. (Lose categorization information)
- Cross reference information

# Commit

- Adds defects to the database
- Remember that we’re not just doing this once.
    - The code might undergo many revisions and have analysis re-run each time
    - Once a bug has been committed we don’t want it reported as a separate entry. (Lose categorization information)

- Cross reference information
- This is where our customers view the defects

# Managing defects

- Prioritize
- Categorize
- Annotate
- Assign
- Integrate with bug tracking systems, lifecycle management systems etc.
- Search by time, file, component, severity etc.
- Set a baseline (no new defects)

# Philosophical

What is a defect anyway?

# Quiz time – what is a defect?

# Quiz time – what is a defect?

- A crash
- Fragile code – unstable under modification
- Doesn’t match the specification
- Doesn’t match the programmer’s intent
- Inconsistent
- Confusing
- Doesn’t obey the house style
- Makes the customer say “What?”
- Something the customer wants to know about
- Inefficient (slow)
- Inefficient (wasteful of some finite resource)
- Security vulnerability
- Non-conformant with an external standard/constraint, such as MISRA

# What is a defect?

```
void foo1(
    int* p,
    int i,
    int j )
{
    if( p )
        foo(p);

    *p = j;
}
```

# What is a defect?

```
void foo1(
    int* p,
    int i,
    int j )
{
    if( p )
        foo(p);

    *p = j;
}
```

```
void foo1a(
    int* p,
    int i,
    int j )
{



    *p = j;
}
```

# What is the defect here?

```
void foo1(
    int* p,
    int i,
    int j )
{
    if( p )
        foo(p);

    *p = j;
}
```

Is the defect:

1. That the pointer was dereferenced without a NULL check?
2. That foo1 checked for a NULL pointer even though it will never be passed a NULL pointer?
3. That the code is inconsistent?

# Programmer’s intent?

```
void nesting_indent_mismatch(int x)
{
    if (x == 0)
        x = foo();
        bar(x);
}
```

# Programmer’s intent?

```
std::string display = foo();

if( condition ) {
    display = bar();
}
else {
    display = baz();
}

write( display );
```

# What does Coverity think of as a defect?

- A crash
- ~~Fragile code – unstable under modification~~
- ~~Doesn’t match the specification~~
- Doesn’t match the programmer’s intent
- Inconsistent
- Confusing **(sort of)**
- ~~Doesn’t obey the house style~~
- Makes the customer say “What?”
- Something the customer wants to know about
- Inefficient (slow)
- Inefficient (wasteful of some finite resource)
- Security vulnerability
- Non-conformant with an external standard/constraint, such as MISRA

# Psychological

Persuasion

# How many people like being told they’re wrong?

# How many people like being told they’re wrong?

The egoless programmer

Jerry Weinberg - *The Psychology of Computer Programming*

# Reporting the problem isn’t enough

Show the evidence

(and even then they might not believe you)

# Bane of our existence #1

| | Coverity does not report bug | Coverity reports bug |
|---|---|---|
| **Bug exists in code** | False Negative | True Positive |
| **Bug does not exist in code** | True Negative | False Positive |

# Bane of our existence #1

| | Coverity does not report bug | Coverity reports bug |
|---|---|---|
| **Bug exists in code** | False Negative | True Positive |
| **Bug does not exist in code** | True Negative | False Positive |

Too many false negatives - our product is ineffective

Too many false positives - loss of confidence

# Bane of our existence #2 - churn

# Bane of our existence #2 - churn

Changes in our results
between versions

# Bane of our existence #2 - churn

- Good churn
  - increase true positive / true negative
  - decrease false positive / false negative

- Bad churn
  - decrease true positive / true negative
  - increase false positive / false negative

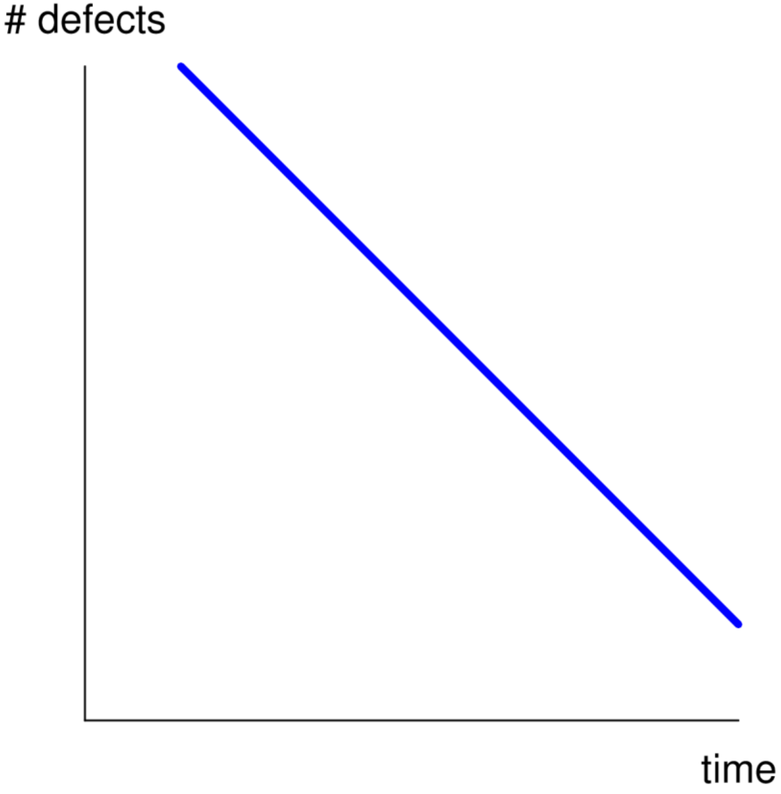
# Bane of our existence #2 - churn

- Good churn
  - increase true positive / true negative
  - decrease false positive / false negative

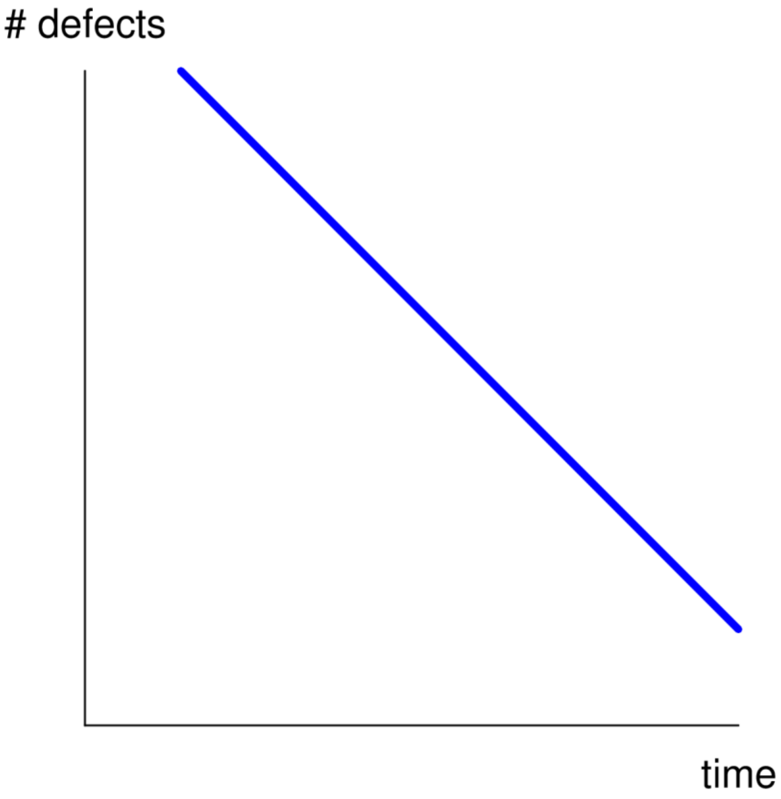
- Bad churn
  - decrease true positive / true negative
  - increase false positive / false negative

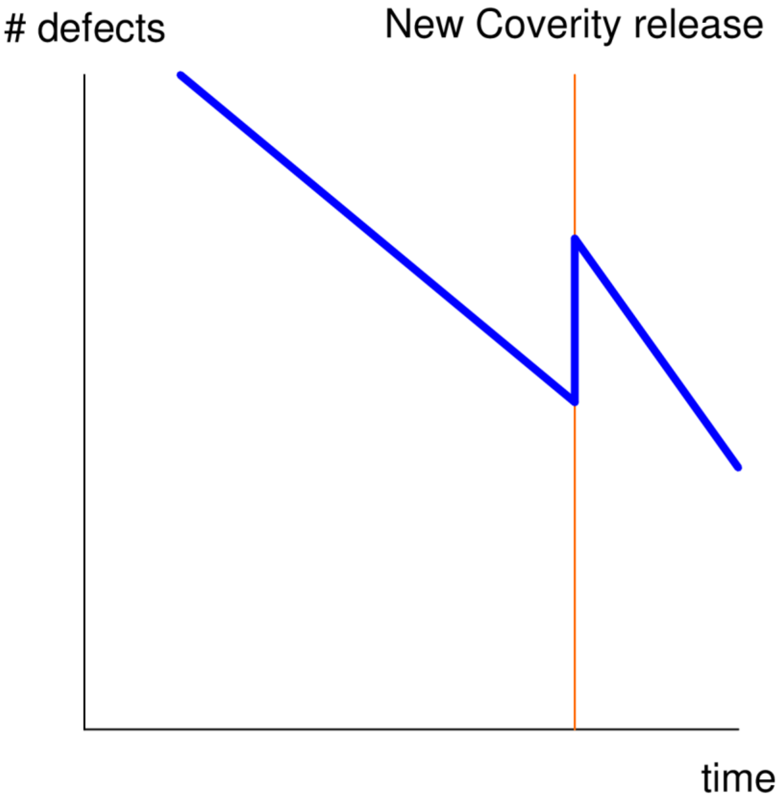
Even good churn can be a problem

# What managers want to see



# What managers actually see





# Bane of our existence #2 - churn

- Managers hate seeing the graph go up
- Can result in customers not upgrading our product (if we can’t see the defect it isn’t there)

# Summary

- Technical - problems are formidable
- Philosophical - defining the problem is tricky
- Psychological - people are always the most complex part of any system

# Thanks

- My Coverity colleagues including:
  - Roger Scott
  - Peter Henriksen
  - Peter Dillinger

**A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World**

coverity®

coverity®