coverity

How to Build Your Own Static Analyzer For Fun and Profit

Dr. Andy Chou
Chief Scientist
Coverity Inc.

Game Plan

coverity

- Market research
- Gather requirements
- Build it
- Sell it
- Take over the world

2

Is it worth doing?

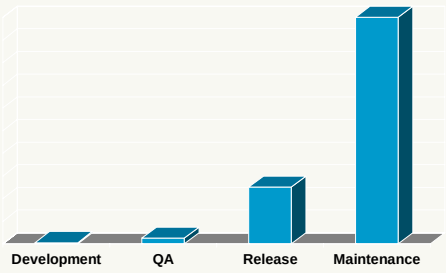
coverity

- NIST Report: bad software costs $59.5 billion/yr
- 0.5% of US GDP
- Ever stayed up all night to find a bug in your code?
- ... someone else's code?

3

Cost of Fixing a Defect

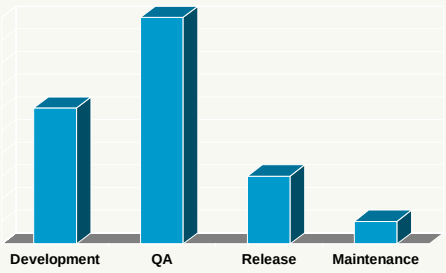
coverity



4

Number of Bugs Found

coverity

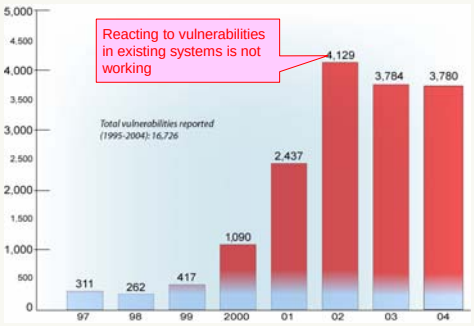


5

Security vulnerabilities

coverity

Source: Robert Seacord, CERT



6

Why Static Analysis?

coverity

- No test cases
- Does not require a "finished product"
- Explores all paths through the code
- Can check a wide variety of program properties
- Applies very early in development cycle

7

Game Plan

coverity

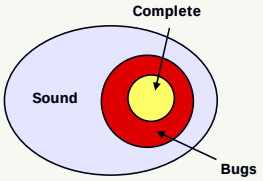
- Market research
- Gather requirements
- Build it
- Sell it
- Take over the world

8

Theoretical Properties

coverity

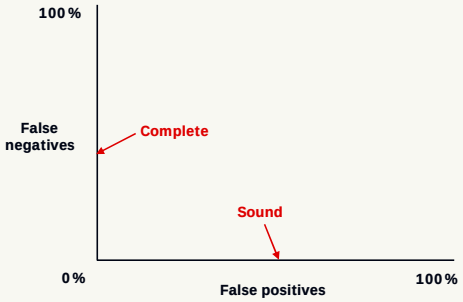
- Soundness
  - No false negatives
- Completeness
  - No false positives



9

Soundness & Completeness

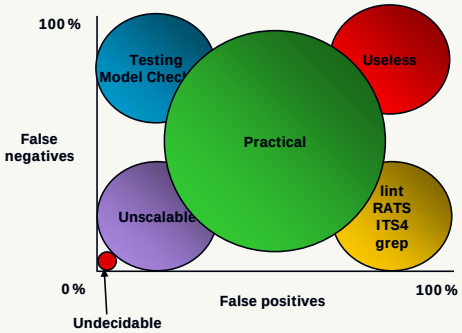
coverity



10

Soundness & Completeness

coverity



11

Soundness & Completeness

coverity

- Neither is necessary nor sufficient to be useful for bug-finding
- Executive decision:
  - Live with some false positives and negatives
  - Bonus round if soundness or completeness can be achieved without sacrificing other requirements

12

Some Real-World Requirements

coverity

- Parse the code
- Find a large number of good bugs
- Few false positives
- Scale to large code bases
- Explain bugs to the user

Two Cardinal Rules

coverity

- The First Two Cardinal Rules of Static Analysis:

  (1) You can't analyze what you can't find.
  (2) You can't analyze what you can't parse.

- Coverity's first year was a crash course in learning these lessons

Parsing Code

coverity

- Myth: The C language exists
- Bigger Myth: The C++ language exists
- What actually exists:
  Compiler = a specific version of a "native" compiler
  f = file.c, plus headers
  o = ordered list of command line options and response files
  e = environment
  c = compiler-specific configuration files

  Lang(Compiler) = { f | ∃ o,e,c Accepts(Compiler,f,o,e,c) }

Parsing Code

coverity

- Need to accept a reasonable subset of:
  $Lang(Compiler_1) \cup Lang(Compiler_2) \cup \ldots$
- Some of the compilers with significant market share:

Parsing Horrors

coverity

- Banal. But take more time than you can believe:

```
void x;
typedef char int;
int foo(int a, int a);
unsigned x @ "TEXT";
int c = L";
unsigned x = 0xdead_beef;
End lines with "\r" rather than "\n"
```

- And, of course, asm:

```
#pragma asm
   mov eax, eab
#pragma end_asm
```

```
asm foo() {
     mov eax, eab;
}
```

```
// newline = end
__asm mov eax, eab

// "]" = end
__asm [
      mov eax, eab
      ]
```


Punting on Parsing

coverity

- Use Edison Design Group (EDG) frontend
- Still need custom rewriter for many supported compilers:

```
205 hpux_compilers.c
215 iar_compiler.c
240 ti_compiler.c
251 green_hills_compiler.c
377 intel_compilers.c
453 diab_compilers.c
```

```
453 sun_compilers.c
485 arm_compilers.c
617 gnu_compilers.c
748 microsoft_compilers.c
1587 metrowerks_compilers.c
...
```

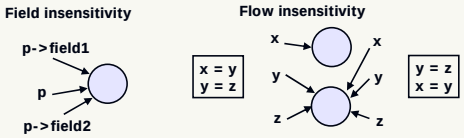
Some Real-World Requirements

- Parse the code
- Find a large number of good bugs
- Few false positives
- Scale to large code bases
- Explain bugs to the user

19

What About Alias Analysis?

- Traditional static analysis for optimizations
- Scalable algorithms are very coarse



20

Bugs to Detect

- Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

21

Open_args

- Prototype for open() syscall:

```
int open(const char *path, int oflag, /* mode_t mode */...);
```

- Typical mistake:

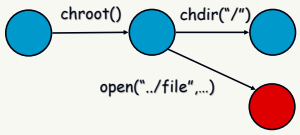
```
fd = open("file", O_CREAT);
```

- Result: file has random permissions
- Check: Look for oflags == O_CREAT without mode argument

22

Chroot checker

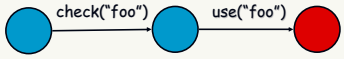
- chroot() changes filesystem root for a process
  - Confine process to a "jail" on the filesystem
- Doesn't change current working directory



23

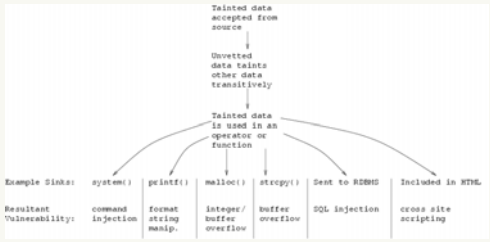
TOCTOU

- Race condition between time of check and use
- Not applicable to all programs



24

Tainting checkers

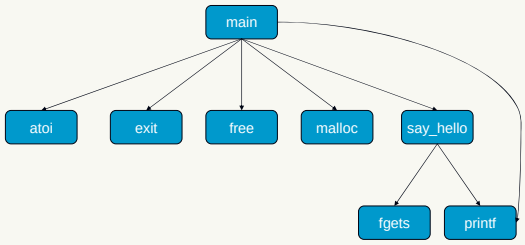


Demo

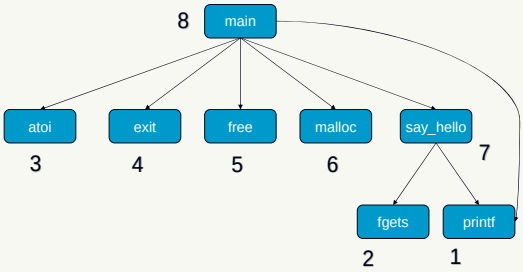
Example Code

```
#include <stdlib.h>
#include <stdio.h>

void say_hello(char * name, int size) {
  printf("Enter your name: ");
  fgets(name, size, stdin);
  printf("Hello %s.\n", name);
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Error, must provide an input buffer size.\n");
    exit(-1);
  }
  int size = atoi(argv[1]);
  char * name = (char*)malloc(size);
  if (name) {
    say_hello(name, size);
    free(name);
  } else {
    printf("Failed to allocate %d bytes.\n", size);
  }
}
```

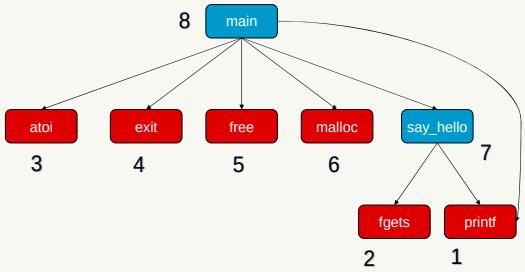

Callgraph



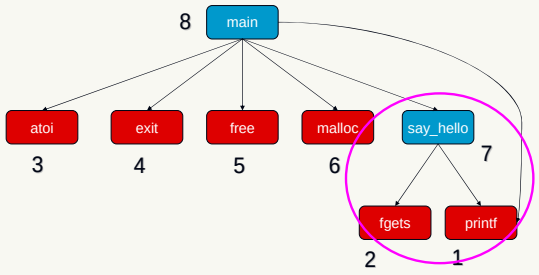
Reverse Topological Sort



Apply Library Models

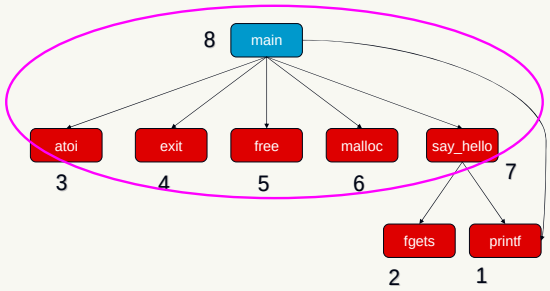


Bottom Up Analysis



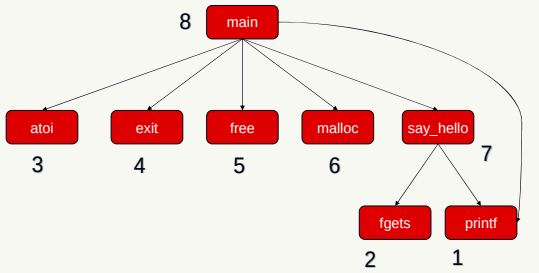
31

Bottom Up Analysis



32

Bottom Up Analysis



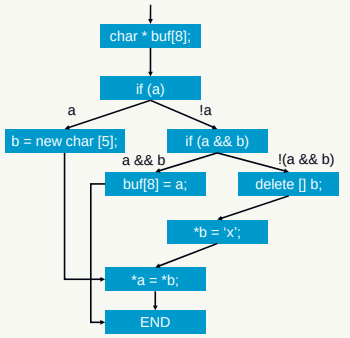
33

Finding Local Bugs

```
#define SIZE 8
void set_a_b(char * a, char * b) {
    char * buf[SIZE];
    if (a) {
        b = new char[5];
    } else {
        if (a && b) {
            buf[SIZE] = a;
            return;
        } else {
            delete [] b;
        }
        *b = 'x';
    }
    *a = *b;
}
```

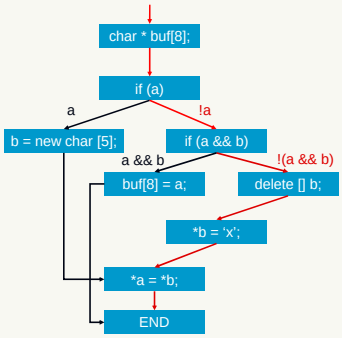
34

Control Flow Graph



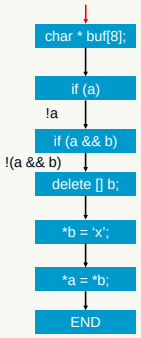
35

Path Traversal



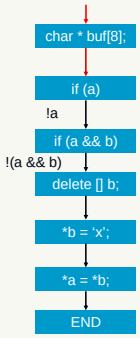
36

Apply Checking



Null pointers | Use after free | Array overrun

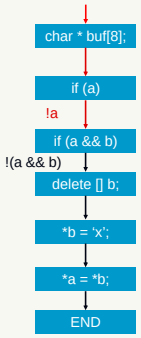
Apply Checking



Null pointers | Use after free | Array overrun

"buf is 8 bytes"

Apply Checking

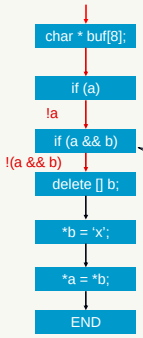


Null pointers | Use after free | Array overrun

"buf is 8 bytes"

"a is null"

Apply Checking

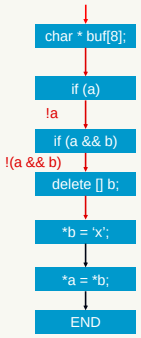


Null pointers | Use after free | Array overrun

"buf is 8 bytes"

"a is null"

Apply Checking



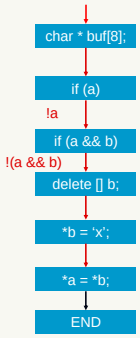
Null pointers | Use after free | Array overrun

"buf is 8 bytes"

"a is null"

"b is deleted"

Apply Checking



Null pointers | Use after free | Array overrun

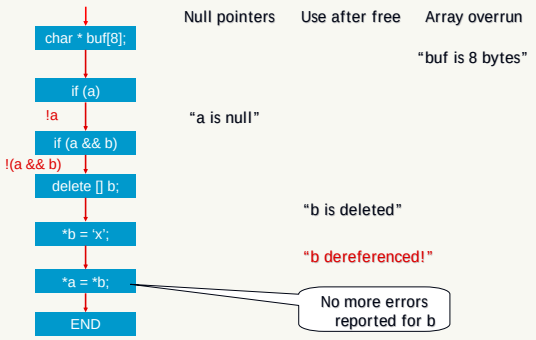
"buf is 8 bytes"

"a is null"

"b is deleted"

"b dereferenced!"

Apply Checking



Some Real-World Requirements

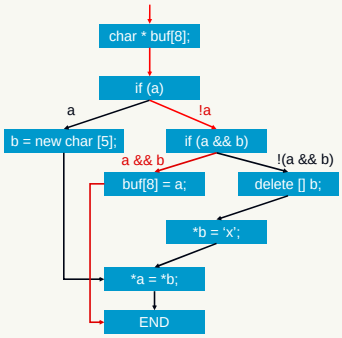
- Parse the code
- Find a large number of good bugs
- Few false positives
- Scale to large code bases
- Explain bugs to the user

False Positives

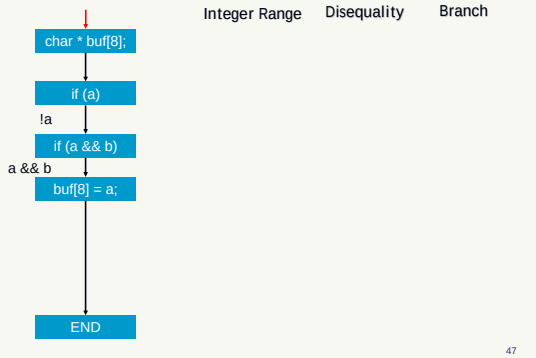
- What is a bug? Something the user will fix.
- Many sources of false positives
  - False paths
  - Idioms
  - Execution environment assumptions
  - Killpaths
  - Conditional compilation
  - "third party code"
  - Analysis imprecision
  - ...

```
p = 0; *p;

fatal("ouch");
```

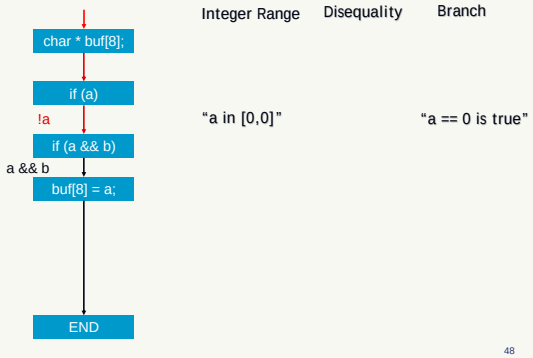

A False Path



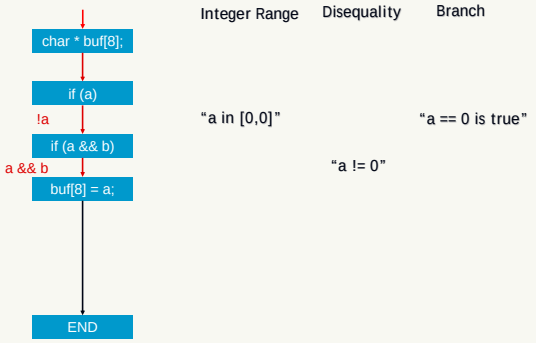
False Path Pruning



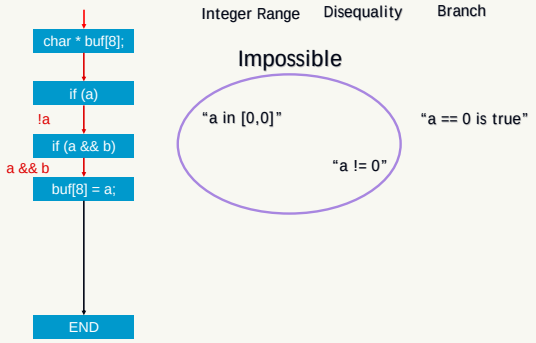
False Path Pruning



False Path Pruning



False Path Pruning



Environment Assumptions

- Should the return value of malloc() be checked?

```
int *p = malloc(sizeof(int));
*p = 42;
```

OS Kernel: Crash machine.	File server: Pause filesystem.	Web application: 200ms downtime
Spreadsheet: Lose unsaved changes.	Game: Annoy user.	IP Phone: Annoy user.
Library: ?	Medical device: malloc?!	

Statistical Analysis

- Assume the code is usually right

3/4 deref

```
int *p = malloc(sizeof(int));
*p = 42;
```

```
int *p = malloc(sizeof(int));
*p = 42;
```

```
int *p = malloc(sizeof(int));
*p = 42;
```

```
int *p = malloc(sizeof(int));
if(p) *p = 42;
```

1/4 deref

```
int *p = malloc(sizeof(int));
if(p) *p = 42;
```

```
int *p = malloc(sizeof(int));
if(p) *p = 42;
```

```
int *p = malloc(sizeof(int));
if(p) *p = 42;
```

```
int *p = malloc(sizeof(int));
*p = 42;
```


Some Real-World Requirements

- Parse the code
- Find a large number of good bugs
- Few false positives
- Scale to large code bases
- Explain bugs to the user

How Large is "Large"?

- Various definitions of what a LOC is
- Some numbers (C/C++ code only)
  - Linux kernel + drivers: ~3.2 MLOC
  - Firefox browser: ~1.9 MLOC
  - Gcc: 0.75 MLOC
  - OpenSSL: 0.21 MLOC
  - XMMS: 0.12 MLOC
- Commercial code bases: 1 MLOC+ is common
- Large sites: 30 MLOC+

Scalability

coverity

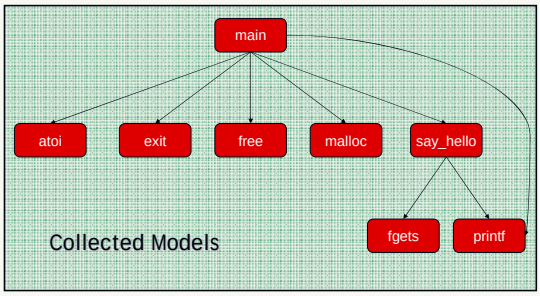
- Usual restriction: nightly build cycle
- Max: ~12 hours (~700 lines/sec)
- Solutions
  - Parallel analysis
  - Incremental analysis
  - Modular analysis

(hard)

55

Modular Analysis

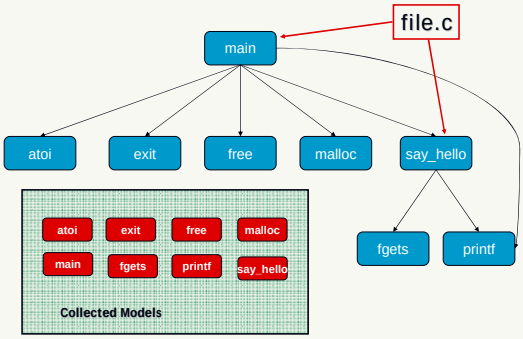
coverity



56

Modular Analysis

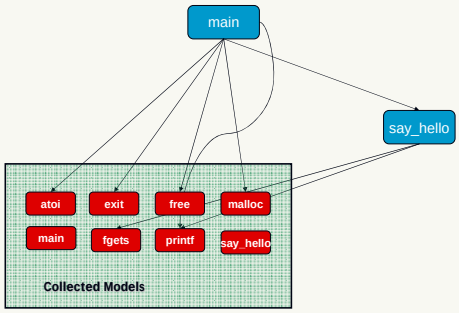
coverity



57

Modular Analysis

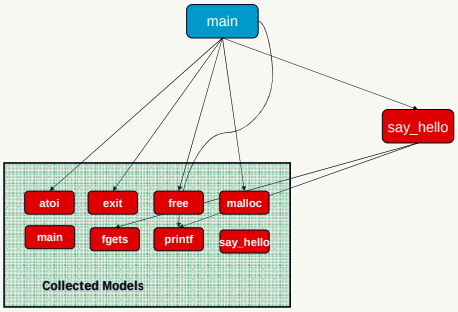
coverity



58

Modular Analysis

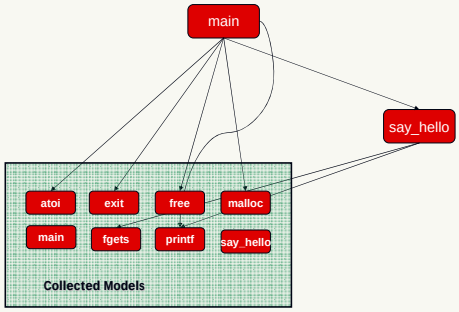
coverity



59

Modular Analysis

coverity



60

Some Real-World Requirements

- Parse the code
- Find a large number of good bugs
- Few false positives
- Scale to large code bases
- Explain bugs to the user

Example from Linux

Example from Linux 2.6.10 (drivers/scsi/aic7xxx_old.c)

```
Added "host" due to comparison "host != 0"
Also see events: [var_deref_model]
At conditional (1): "host != 0" taking false path
8427      if (host != NULL)
8428      {
8429        p = (struct aic7xxx_host *) host->hostdata;
8430        memset(p, 0, sizeof(struct aic7xxx_host));
8431        *p = *temp;
8432        p->host = host;
8433
8434        p->scb_data = kmalloc(sizeof(scb_data_type), GFP_ATOMIC);
8435        if (p->scb_data != NULL)
8436        {
8437          memset(p->scb_data, 0, sizeof(scb_data_type));
8438          scbq_init (&p->scb_data->free_scbs);
8439        }
8440        else
8441        {
8442          /*
8443           * For some reason we don't have enough memory.  Free the
8444           * allocated memory for the aic7xxx_host struct, and return NULL.
8445           */
8446          release_region(p->base, MAXREG - MINREG);
8447          scsi_unregister(host);
8448          return(NULL);
8449        }
8450        p->host_no = host->host_no;
8451      }
Variable "host" tracked as NULL was passed to a function that dereferences it. [model]
Also see events: [var_compare_op]
8452      scsi_set_device(host, &p->pdev->dev);
8453      return (p);
8454    }
8455
```


Explaining Bugs

```
Added "host" due to comparison "host != 0"
Also see events: [var_deref_model]
At conditional (1): "host != 0" taking false path
8427      if (host != NULL)
8428      {
```

- The red messages are *events*, which are the places in the code where the analysis makes a decision
- The green messages indicate the path the analysis followed
- In this case, the analysis indicates that when following the false path from the comparison on **line 8427**, the variable **host** is **NULL**

Explaining Interprocedural Bugs

```
Variable "host" tracked as NULL was passed to a function that dereferences it. [model]
Also see events: [var_compare_op]
8452      scsi_set_device(host, &p->pdev->dev);
8453      return (p);
8454    }
8455
```

- When the analysis reports a message that crosses procedure boundaries, a "model" link appears
  - Following the model link will show the event in the callee function
- Each variable, field, function call is linked to its declaration through the cross referencing links

Explaining Interprocedural Bugs

Potentially **NULL** pointer

```
542     static inline void scsi_set_device(struct Scsi_Host *shost,
543                                        struct device *dev)
544     {
Directly dereferenced parameter "shost"
545             shost->shost_gendev.parent = dev;
546     }
```


Game Plan

- Market research
- Gather requirements
- Build it
- Sell it
- Take over the world

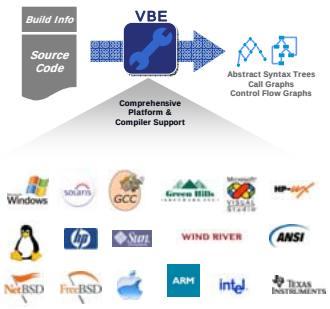
Coverity's Commercial History

Breakthrough technology out of Stanford	Company incorporated	Achieved profitability	Product growth and proliferation
2000-2002	2002	2003	2004-05
• Meta-level compilation checker ("Stanford Checker") detects 2000+ bugs in Linux.	• Deluge of requests from companies wanting access to the new technology. • First customer signs: Sanera systems	• 7 early adopter customers, including VMWare, SUN, Handspring. • Coverity achieves profitability.	• Version 2.0 product released. • Company quadruples • 110+ customers including Juniper, Synopsys, Veritas, nVidia, palmOne. • Self funded

Stanford

67

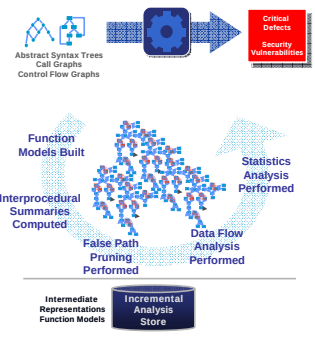
VBE (Virtual Build Environment)



No changes required to your build environment

- Utilizes automated configuration
- Leverages existing build system
- Builds detailed and accurate representation of whole code
  - Abstract syntax trees
  - Call graphs
  - Control flow graphs

Analysis Engine



Deep and comprehensive Analysis and Detection

- Sophisticated dataflow analysis
  - Cross-functional
  - Context aware
  - Value tracking
- 100% of all possible paths
- Deeper inspection
  - Uncovers more defects
  - Better accuracy
  - 20% false positive rate
- Scalable to tens of millions of lines of code
- Unparalleled performance
  - Subsequent runs
  - No loss of accuracy
  - Full analysis on desktop

Prevent Library of Checkers

Quality	Concurrency	Security
• System and Process Crash • Memory/Resource Leaks • Data, Memory, File Corruption • Performance Degradation • Unpredictable Behavior	• Deadlocks • Lock Contention • Unpredictable performance • Performance degradation	• Denial of Service • Privilege Escalation • Malicious Code

Checker Library

Resource Problems
- Resource Leak
  - Memory leak
  - File pointer leak
  - System resource leak

Concurrency Problems
- Double Lock
- Missing unlock
- Incorrect lock acquisition
- Sleeping while locked

API Usage Errors
- Passing large parameters
- Insecure temp file creation
- Improper method override

Pointer Errors
- Use of uninitialized data
  - Uninitialized memory
  - Uninitialized variable
- Mismatched allocation operators
- Dereferencing invalid pointers
  - Null pointer dereference
  - Wrong address space
  - Accessing freed pointers
- Dangling stack references
- Use of freed resource
  - Double free (memory, file pointers, system resources)
  - Use after free (memory, file pointers, system resources)

Logic Errors
- Flawed branch logic
- Use of invalid STL iterators
- Useless operation
- Inconsistent error handling
- Security logic errors
  - Time of check, time of use
  - Insecure file creation
  - Improper chroot
  - Improper privilege inheritance

Security Warnings
- Potentially insecure coding practices

Bounds Errors
- Out of bounds array access
- Buffer underflow
- Stack smashing
  - Stack overflow
  - Stack buffer overrun
  - Stack string overrun
- Bad negative integer cast
- Incorrect allocation size
- Non-null terminated strings

External Data Handling
- Integers
  - Loop bound
  - Array access
  - Allocation size
- Strings
  - Buffer overflow
  - SQL Injection
  - Format string errors
  - Cross-site scripting

70

Game Plan

- Market research
- Gather requirements
- Build it
- Sell it
- Take over the world

71

Sales Model

- Offer prospects a free trial
- Run over tons of code, find tons of bugs
- Have a meeting showing bugs to developers and their managers
- Delete most of the results and go home
- Wait for them to call back

72

Academics don't understand money

- "We'll just charge per seat like everyone else"
  - Finish the story: "Company X buys three Purify seats, one for Asia, one for Europe and one for the US…"
- Try #2: "we'll charge per lines of code"
  - "That is a really stupid idea: (1) …, (2) … , … (n) …"
  - Actually works. I'm still in shock. Would recommend it.
- Good feature for seller:
  - No seat games. Revenue grows with code size. Run on another code base = new sale.
- Good feature for buyer: No seat-model problems
  - Buy once for project, then done. No per-seat or per-usage cost; no node lock problems; no problems adding, removing or renaming developers (or machines)
  - People actually seem to like this pitch.

73

Some experience

- Surprise: Sales guys are great
  - Easy to evaluate. Modular.
- Company X buys tool, then downsizes. Good or bad?
  - For sales, very good: X fires 110 people. They get jobs elsewhere. Recommend coverity. 4 closed deals.
- Large companies "want" to be honest
  - Veritas: want monitoring so don't accidently violate!
- What can you sell?
  - User not same as tool builder. Naïve. Inattentive. Cruel.
  - Makes it difficult to deploy anything sophisticated.
  - Example: statistical inference.
  - Some ways, checkers lag much behind our research ones.

74

"No, your tool is broken: that's not a bug"

- "No, the loop will go through once!"

```
for(i=1; i < 0; i++) {
  …deadcode…
}
```

```
for(s=0; s < n; s++) {
  …
  switch(s) {
  case 0: assert(0); return;
  …
  }
  …dead code…
```

- "No, && is 'or'!"

```
void *foo(void *p, void *q) {
  if(!p && !q)
    return 0;
```

- "No, ANSI lets you write 1 past end of the array!"
  - ("We'll have to agree to disagree." !!!!)

```
unsigned p[4];   p[4] = 1;
```

75

Game Plan

- Market research
- Gather requirements
- Build it
- Sell it
- Take over the world

76

A Partial List of 110+ Customers

EDA: Synopsys, Mentor Graphics, Cadence

Storage: Veritas, EMC, Panasas

Security: McAfee, PGP, Check Point

Networking: Juniper, Avici

Government: NASA, National Aeronautics and Space Administration, Jet Propulsion Laboratory

Embedded: Qualcomm, Symbol, Keithley

Biz Applications: Oracle, bmcsoftware

OS: Wind River, palmOne

Open Source: MySQL, mozilla, Apache

77

Open Source Quality Project

Vulnerability Discovery and Remediation Open Source Hardening Project

Purpose

- Research latest source analysis techniques for security and quality
- Establish new baseline for software quality and security in open source
- Establish a metric for software quality

78

Slide 73

DE1 whole bunch of options: razor blade model, where we give away checkers for free and charge for system. or inverse razor where we give away system and charge for checkers. or charge per seat, or charge per lines of code (prefix). get a lot of pushback on the last one. prefix worked ok, but not what we would consider a success.

i argued very strongly against per line model. completely wrong.
Dawson Engler, 8/23/2005

Slide 74

DE2 count how often something true versus not. sort in decreasing deviance. inspect until hit fp. developer inspect all, mark as FP. say tool sucks to everyone.
Dawson Engler, 8/23/2005

Slide 75

DE3 we know about these since happen in results meetings. a bit dangerous, but usually other developers will laugh at the confused one. scary to think of other times when things just marked as FP.
Dawson Engler, 8/23/2005

Preliminary Findings

coverity

- Over 50 commonly used open source packages
- Over 32 MLOC analyzed nightly with 3 standard boxes
- More packages added regularly

- .434 defects/KLOC open source average
- .290 defects/KLOC LAMP stack

79

FreeBSD

coverity

"Coverity's Prevent is an invaluable tool that we've now been able to integrate into the FreeBSD Project development process with nightly source code scans.

Eighty-five Free BSD developers are now registered to review Coverity-generated bug reports, resulting in hundreds of important bug fixes, one leading to a security advisory.

Coverity's contributions have significantly improved the quality of Free BSD source codebase, which is greatly appreciated by both FreeBSD developers and users"

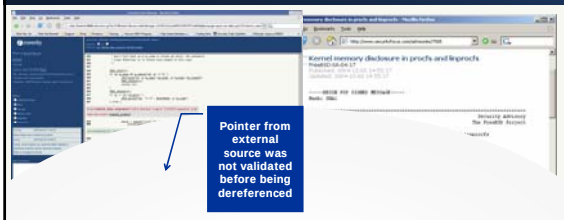
-- Robert Watson,
President FreeBSD Foundation

- 1,682,166 LOC
- 635 defects (.377 defects /KLOC)

80

Free BSD Security Alert

coverity



Malicious local user could perform a local denial of service attack by causing a system panic. A user could read parts of kernel memory such as the file cache or terminal buffers which can include user-entered passwords.

X.Org Case study

coverity

X.Org Foundation

- X Windows Systems (also known as X11)
- Originated at MIT in 1984
- Open source
- Primary GUI for Unix operating systems including Linux, Solaris and others
- Bundled in Open VMS and Mac OS X and many other operating systems environments

82

X.Org Case study

coverity

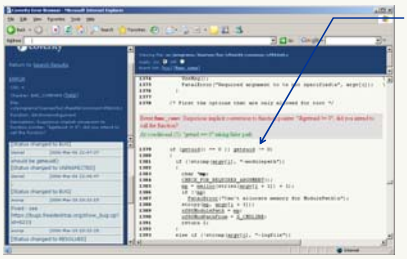
X.Org Foundation

- Scan found 1681 potential defects in 2.3 MLOC
- Defect rate is .71 defects /KLOC
  - Above average
- First 11 days inspected 879 with an out-of-the-box false positive rate of 21%
- Half way through fixing reported bugs

83

Example Security Violation (Bug #4)

coverity



Missing () to libc function geteuid detected by BAD_COMPARE checker

Since without the parentheses, the code is simply checking to see if the geteuid function in libc was loaded somewhere other than address 0 (which is pretty much guaranteed to be true), it was reporting it was safe to allow risky options for all users, and thus a security hole was born.

- Alan Coopersmith, Sun Developer

X.Org Privilege Escalation

coverity

```
/* First the options that are only allowed for root */
 if (getuid() == 0 || geteuid != 0)
 ...
```


X.Org Privilege Escalation

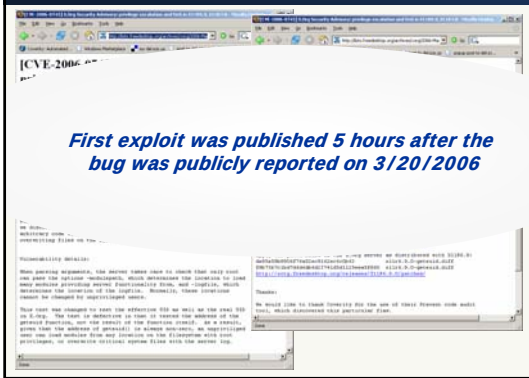
coverity

```
/* First the options that are only allowed for root */
 if (getuid() == 0 || geteuid() != 0)
 ...
```

- The vulnerability allows ordinary users on a system to perform the following
  - Run arbitrary code as root administrator
  - Create and overwrite files as root administrator

Security Advisory

coverity



First exploit was published 5 hours after the bug was publicly reported on 3/20/2006

Ethereal 0.99.0

coverity

- Released 4/24/06

Under a grant funded by the U.S. Department of Homeland Security, Coverity has uncovered a number of vulnerabilities in Ethereal:
The statistics counter could crash Ethereal. Versions affected: 0.10.10 - 0.10.14. CVE: CVE-2006-1937
Ethereal could crash while reading a malformed Sniffer capture. Versions affected: 0.8.12 - 0.10.14. CVE: CVE-2006-1938
An invalid display filter could crash Ethereal. Versions affected: 0.9.16 - 0.10.14. CVE: CVE-2006-1939
The general packet dissector could crash Ethereal. Versions affected: 0.10.9 - 0.10.14. CVE: CVE-2006-1937
The AIM dissector could crash Ethereal. Versions affected: 0.10.7 - 0.10.14. CVE: CVE-2006-1937
The RPC dissector could crash Ethereal. Versions affected: 0.9.8 - 0.10.14. CVE: CVE-2006-1932
The DCERPC dissector could crash Ethereal. Versions affected: 0.9.16 - 0.10.14. CVE: CVE-2006-1939
The ASN.1 dissector could crash Ethereal. Versions affected: 0.9.8 - 0.10.14. CVE: CVE-2006-1939
The SMB PIPE dissector could crash Ethereal. Versions affected: 0.8.20 - 0.10.14. CVE: CVE-2006-1938
The BER dissector could loop excessively. Versions affected: 0.10.4 - 0.10.14. CVE: CVE-2006-1933
The SNDCP dissector could abort. Versions affected: 0.10.4 - 0.10.14. CVE: CVE-2006-1940
The Network Instruments file code could overrun a buffer. Versions affected: 0.10.0 - 0.10.14. CVE: CVE-2006-1934
The NetXray/Windows Sniffer file code could overrun a buffer. Versions affected: 0.10.13 - 0.10.14. CVE: CVE-2006-1934
The GSM SMS dissector could crash Ethereal. Versions affected: 0.9.16 - 0.10.14. CVE: CVE-2006-1939
The ALCAP dissector could overrun a buffer. Versions affected: 0.10.14. CVE: CVE-2006-1934
The telnet dissector could overrun a buffer. Versions affected: 0.8.5 - 0.10.14. CVE: CVE-2006-1936
ASN.1-based dissectors could crash Ethereal. Versions affected: 0.9.10 - 0.10.14. CVE: CVE-2006-1939
The H.248 dissector could crash Ethereal. Versions affected: 0.10.11 - 0.10.14. CVE: CVE-2006-1937
The DCERPC NT dissector could crash Ethereal. Versions affected: 0.9.14 - 0.10.14. CVE: CVE-2006-1939
The PER dissector could crash Ethereal. Versions affected: 0.9.14 - 0.10.14. CVE: CVE-2006-1939
Under Windows, Unicode characters in profile and configuration file paths could cause problems. Versions affected: 0.10.14.
The Coverity audit turned up several UI-related bugs that could make Ethereal crash.

More Experiences

coverity

- Security is dismal in the real world
- For many companies, source code analysis is still #4 or #5 on the priority list
  - Patching process (get customers to react to the problem)
  - Marketing (convince the customer there is no problem)
  - Training (teach customers how to turn off the parts with problems)
  - Tiger teams (guess what the problems might be)

"There are a thousand hacking at the branches of evil to one who is striking at the root"

- Henry David Thoreau, *Walden*, 1854

Conclusion

coverity

- Bugs are everywhere
- No bug is too stupid to look for
- Finding errors often easy, saying why is hard
- False positives kill usage, credibility, ROI
- Static analysis will become mainstream

Backup slides

coverity

91

Myth: More analysis is always better

coverity

- Does not always improve results, and can make worse
- The best error:
  - Easy to diagnose
  - True error
- More analysis used, the worse it is for both
  - More analysis = the harder error is to reason about, since user has to manually emulate each analysis step.
  - Number of steps increase, so does the chance that one went wrong. No analysis = no mistake.
- In practice:
  - Demote errors based on how much analysis required
  - Revert to weaker analysis to cherry pick easy bugs
  - Give up on errors that are too hard to diagnose.

92

Myth: Soundness is a virtue

coverity

- Soundness: Find all bugs of type X.
  - Not a bad thing. More bugs good.
  - BUT: can only do if you check weak properties.
- What soundness really wants to be when it grows up:
  - Total correctness: Find all bugs.
  - Most direct approximation: find as many bugs as possible.
- Opportunity cost:
  - Diminishing returns: Initial analysis finds most bugs
  - Spend on what gets the next biggest set of bugs
  - Easy experiment: bug counts for sound vs unsound tools.
- End-to-end argument:
  - "It generally does not make much sense to reduce the residual error rate of one system component (property) much below that of the others."

93