Static Analysis Use Case
Samba and Coverity
David Maxwell for Scale 9x
Coverity Scan

• Launched, March 2006

• DHS sponsored “Open Source Hardening Project”
  – 2006-2009

• Using Coverity’s commercial static analysis product to identify bugs at the source code level

• 35 open source projects on day one

• Since grown to 300+ projects

• Over 15,000 bugs fixed
There is no single measure of the effectiveness of a tool on the software development process.
Since we can never run the same development effort twice, with identical teams, portions of this evaluation are highly subjective.
Evaluating Effectiveness

- **Objective measures**
  - Static Analysis produced defect counts
  - Numbers of Bug Reports
  - Defects confirmed as ‘real’ by the developers

- **Subjective measures**
  - Anecdotal comments by developers
  - Community feedback
  - ‘Support Load’ reduction
Objective measures

• Static Analysis produced defect counts
  – Good objective measure
    • Reproducible
    • Consistent
    • Low effort to collect
    • Automatable
    • “Static Analysis Tools as Early Indicators of Pre-Release Defect Density” - Microsoft Research Paper
Objective measures

- Numbers of Bug Reports
  - Potentially useful if all other factors are controlled
  - Not the case in our example
    - Multiple development branches
    - Concurrent new development during defect resolution
    - Userbase changes over time
    - Platform support changes over time
Objective measures

• Defects confirmed as ‘real’ by the developers
  – A high False Positive rate would bring the defect count metric into question
  – Would also affect future developer trust in the analysis tool
Subjective measures

- Anecdotal comments by developers
  - Informative, but not comparable between projects

- Community feedback
  - Dependent on the nature of each project’s community
Subjective measures

• ‘Support Load’ reduction
  – Difficult to quantify in an open source environment, due to the variety of support channels
Your Measures?

As in most engineering problems...

What do you want to minimize?

– Immediate Cost
– Long Term Cost
– Time
– Manpower
– Ongoing Support
• **Samba**
  – Open source networking suite
  – Provides Microsoft protocol compatibility
  – International team, started in Australia
  – Project founded in 1992
  – ~300KLOC -> 850KLOC 2006-now
Use Case – Samba & Coverity

- Started regular scanning March 2006
- 14 Developers accessing the results
- Database available 24/7, SAAS
- New analysis every 2 days on average
  - (797 builds in database)
Use Case – Samba & Coverity

- Static Analysis defect counts, 310KLOC
Use Case – Samba & Coverity

Day 1: Fixed

- 4 NULL Pointer derefs
- 10 Resource leaks
- 1 Uninitialized data
- 31 Use after free

But – other changes that day introduced new defects

![Defect Count Chart]

- 203 defects on 3/6/2006
- 197 defects on 3/7/2006
- 113 defects on 3/9/2006
- 99 defects on 3/10/2006
- 62 defects on 3/12/2006
- 29 defects on 3/13/2006
- 14 defects on 3/14/2006
- 0 defects on 3/15/2006

Defect Count
Use Case – Samba & Coverity

Day 2: Fixed

- 15 NULL Pointer derefs
- 4 Resource leaks
- 1 static buffer overrun
- 53 Use after free
- 3 returned NULL
- 2 bad comparison
- 1 Dead code

Defect Count

- 203 on 3/6/2006
- 197 on 3/7/2006
- 113 on 3/9/2006
- 99 on 3/10/2006
- 62 on 3/12/2006
- 29 on 3/13/2006
- 14 on 3/14/2006
- 0 on 3/15/2006

Defect Count
if (!brl_lock) {
    return False;
}
Event **func_conv**: Suspicious implicit conversion to function pointer:
"&brl_lock == 0";

did you intend to call the function?

```c
118     if (!brl_lock) {
119             return False;
120         }
```
/***************************************************************************/
Lock a range of bytes.***************************************************************************/

NTSTATUS brl_lock(struct byte_range_lock *br_lck,
uint16 smbpid,
struct process_id pid,
br_off start,
br_off size,
enum brl_type lock_type,
enum brl_flavour lock_flav,
BOOL *my_lock_ctx)
{
NTSTATUS ret;
struct lock_struct lock;

*my_lock_ctx = False;
Use Case – Samba & Coverity

Defect Count

Would this graph be solid blue?
Use Case – Samba & Coverity

Defect Count

Would this graph be solid blue?
Use Case – Samba & Coverity

- Defects confirmed as ‘real’ by the developers

  13 defects marked False Positive
  216 total defects

  \[
  \frac{13}{216} = 6\%
  \]
Use Case – Samba & Coverity

• Subjective measures
  – Anecdotal comments by developers

“This tool has become part of our process”
“Using […] source code analysis technology is like having a developer on the team with an inhuman attention to detail, who points out all the corner cases and boundary conditions developers didn’t consider when they first wrote the code.”
“I code more carefully, because I know my laziness will be caught and embarrass me.”
• Community feedback
Use Case – Samba & Coverity

- Community feedback
  - Invited to give opening keynote at annual Samba conference in 2009
Open Source Reports

  - Open Source Report 2008
  - Open Source Report 2009
  - Open Source Report 2010 (Android & Supply Chain)

Most Commonly Found Defects

Looking at the most commonly found defects in code, we found three classes of defects that are often found in open source software:

1. **NULL Pointer Divergence**
2. **Inconsistent Locks**
3. **Use Before Test (NULFs)**
4. **Use After Free**
5. **Rathak Oversons (additionally allocated)**
6. **Usable Use of Reserved NULL**
7. **Uninitialized Values (null)**
8. **Use of Reserved Negative**
9. **Type and Allocation Size Mismatch**
10. **Rathak Oversons (dynamically allocated)**

The graph shows the frequency of functions by length and the frequency distribution is skewed to the left due to errors in the code. This indicates that most defects are found in functions that are shorter in length. The histogram shows a clear trend of a larger number of defects in shorter functions. The top 10 defects are responsible for a significant portion of the defects found in the codebase. The distribution of defects is not evenly spread across all lengths, with a higher concentration in shorter functions. This suggests that shorter functions are more prone to defects, which can be attributed to various factors such as haste, lack of testing, or inadequate coding practices. The analysis highlights the importance of focusing on shorter functions to improve the overall quality of the codebase.
• Questions?