Finding Bugs is Easy

David Hovemeyer and William Pugh

October 27, 2004
Finding Bugs is Easy

Introduction

Bug Patterns

Evaluation

Conclusions
Bugs in software

- Programmers are smart
- We have good techniques (e.g., unit testing, pair programming, code inspections) for finding bugs early
- So, most bugs remaining in production code must be subtle, and require sophisticated techniques to find
- Right?
Apache Ant 1.6.2,
org.apache.tools.ant.taskdefs.optional.metamata.MAudit

if (out == null) {
    try {
        out.close();
    } catch (IOException e) {
    }
}
Eclipse 3.0.1, org.eclipse.update.internal.core.ConfiguredSite

if (in == null)
    try {
        in.close();
    } catch (IOException e1) {
    }
}
• JBoss 4.0.0RC1, org.jboss.mq.xml.XElement
  if ( split[0].equals( null ) ) {
    return this;
  }

• JBoss 4.0.0RC1, org.jboss.cache.TreeCache
  int treeNodeSize=fqn.size();
  if(fqn == null) return null;
J2SE version 1.5 build 63 (released version),
java.lang.annotation.AnnotationTypeMismatchException

public String foundType() {
    return this.foundType();
}
Software contains bugs

- Lots of obvious bugs find their way into production software
- Testing and code inspections won’t find every bug
  - Very hard to get high test coverage for a large system
  - Limits to frequency, completeness of code inspections
- Techniques to find more bugs automatically are valuable
Let the computer figure out where (some of) the bugs are.

Much work has been done on static analysis to find bugs:
- Lint, PREfix, PREfast, FxCop, MC/Metal, ESC/Java, Cqual
- Many other tools, papers, techniques

However, static bug checking not used nearly as widely as testing, code inspections:
- We think it should be!
Static analysis challenges

- Fundamental limits to static analysis
  - Nontrivial properties of programs are undecidable
- Choice: what to do when confronted by a difficult analysis problem
  - Be consistently conservative: could choose to
    - Never miss a real bug (but report some false positives)
    - Never report a false positive (but miss some real bugs)
  - Guess a “likely” behavior
    - Both false positives and false negatives are possible
    - But, may be able to get better accuracy overall
Bugs vs. style

- It is important to distinguish *bug checkers* from *style checkers*
  - Style checkers warn about dubious or dangerous coding idioms: however, instances of those idioms may not be particularly likely to be a bug
  - Bug checkers warn about code idioms that are likely to be actual bugs
- Style checkers useful for enforcing consistent coding standards
  - Help *prevent* certain kinds of bugs
Introduction

Bug Patterns

Evaluation

Conclusions
Bug-driven bug finding

- When a bug is found, a good developer will:
  - Fix it
  - Add a dynamic check or test case to make sure the bug cannot reoccur
- Why not take this idea a step further?
  - Write a static bug checker to find occurrences of similar bugs elsewhere in the program, or in other programs
Bug patterns

- Many bugs share common characteristics
- These common characteristics form *bug patterns*
- Can often be detected using simple analysis techniques
- Consequences of imprecision:
  - May miss some real bugs
  - May report *false warnings*
Bug arose in intro programming course

```java
class WebSpider {
    /** Construct a new WebSpider */
    public WebSpider(boolean isDFS, int limit) {
        WebSpider spider = new WebSpider(isDFS, limit);
    }
}
```

Bug is unconditional self-recursive invocation: infinite loop
- Checked, 4 other students had similar bugs

Wrote detector to find unconditional self-recursion; ran it on JDK1.5 rt.jar to ensure it wasn’t generating false positives
- Found 3 real bugs!
Two approaches to devising a static analysis to find bugs:

1. Given an analysis technique, figure out what bugs it could find
2. Given a bug, figure out an analysis that could be used to find occurrences of similar bugs
We have implemented automatic detectors for about 50 bug patterns in a tool called FindBugs

- Open source

Analyzes Java bytecode using Apache BCEL library

- Bytecode is easy to analyze
- Tool continues to work in the face of language changes (e.g., new Java 1.5 language features)
Implementing a bug pattern detector

- Implementation steps:
  1. Think of the simplest technique that would find occurrences of the bug
  2. Implement it
  3. Apply it to real software
     - Hopefully find some real bugs
     - Will probably produce some false warnings
  4. Add heuristics to reduce percentage of false warnings

- Our experience: new detectors can usually be implemented quickly (somewhere between a few minutes and a few days)
- Often, detectors find more bugs than you would expect
Implementation techniques

- We use various kinds of analysis in implementing detectors:
  - Examination of method names, signatures, class hierarchy
  - Linear scan of bytecode instructions using a state machine
  - Method control flow graphs, dataflow analysis
- No interprocedural flow analysis or sophisticated heap analysis
Categories of bug patterns

- Correctness
  - Multithreaded correctness
- Malicious code vulnerability
- Efficiency and design

Will we describe a few of the patterns and show some examples of bugs found

- See paper, website for more bug patterns
Correctness bugs
Equal objects must have equal hash codes
- Programmers sometimes override equals() but not hashCode()
  - Or, override hashCode() but not equals()
- Objects violating the contract won’t work in hash tables, maps, sets

Example (JDK 1.5 build 59):
- javax.management.Attribute

Warnings: 55 in rt.jar 1.5-b59, 170 in eclipse-3.0
Covariant Equals

- equals() method must have parameter of type Object
- Programmers sometimes define with the type of the class
  - Doesn’t actually override Object.equals()
  - The right equals() won’t get used in Collections
- Examples (JDK 1.5 build 59)
  - java.awt.geom.Area
  - sun.security.krb5.Realm
- Warnings: 9 in rt.jar 1.5-b59, 3 in eclipse-3.0
Null Pointer Dereference

- Often, these happen because of trivial mistakes (e.g., using && instead of ||, or vice versa)
- Sometimes, code is modified incorrectly during maintenance
- Bugs: 37 in rt.jar 1.5-b59, 55 in eclipse-3.0
Null pointer examples

- Eclipse 3.0.1, org.eclipse.team.internal.ccvs.core.CVSSyncInfo
  ```java
  if (local != null || local.getType() == IResource.FILE) {
  ```

- Eclipse 3.0.1, org.eclipse.debug.internal.ui.sourcelookup.AddSourceContainerDialog
  ```java
  if (browser == null) {
    super.okPressed();
  }
  ISourceContainer results =
    browser.addSourceContainers(getShell(), fDirector);
  ```
Null pointer examples

- JBoss 4.0.0RC1, javax.xml.soap.SOAPPart, getContentId()
  
  ```java
  if( header != null || header.length > 0 )
      id = header[0];
  ```

- JBoss 4.0.0RC1, javax.xml.soap.SOAPPart, getContentLocation()
  
  ```java
  if( header != null || header.length > 0 )
      location = header[0];
  ```
Return value ignored

- Many API methods can only be used correctly if return value is checked
  - E.g., methods that perform an operation on an immutable object such as a String
  - Programmers might think operation actually modifies object
- Bugs: 5 in rt.jar 1.5-b59, 7 in eclipse-3.0
Return value ignored examples

- Eclipse 3.0.1, org.eclipse.ui.externaltools.internal.model.BuilderUtils

  String name = workingCopy.getName();
  name.replace('/', '.');
  if (name.charAt(0) == ('.')) {

  


Return value ignored examples

- Eclipse 3.0.1,
  org.eclipse.update.internal.ui.security.UpdateManagerAuthenticator

  String hostString = host;
  if (hostString == null && address != null) {
      address.getHostString();  // Meant to assign to hostString?
  }
  if (hostString == null) {
      hostString = "";  //NON-NLS-1$
  }

Multithreaded correctness bugs
Inconsistent synchronization

- Detecting data races: NP hard in general case
  - Many complicated analyses have been developed to find data races
  - What if we try looking for very obvious data races?
- Common idiom for thread safe classes is to synchronize on the receiver object ("this")
- Examine all field accesses and synchronized regions
  - Find fields where lock on "this" object is sometimes, but not always, held
  - Unsynchronized accesses, if reachable by multiple threads, constitute a potential data race
- Bugs: 52 in rt.jar 1.5-b59, 39 in eclipse-3.0
Inconsistent synchronization example

- GNU Classpath 0.08, java.util.Vector

```java
public int lastIndexOf(Object elem)
{
    return lastIndexOf(elem, elementCount - 1);
}
```

```java
public synchronized int lastIndexOf(Object e, int index)
{
    ...
```
Unconditional wait

- Before waiting on a monitor, the condition waited for should be checked
  - Waiting unconditionally upon entering a synchronized block usually a bug
  - If condition checked without lock held, could miss notification
- Bugs: 3 in rt.jar 1.5-b59, 2 in eclipse-3.0
Unconditional wait example

- JBoss 4.0.0RC1, org.jboss.deployment.scanner.AbstractDeploymentScanner
  
  // If we are not enabled, then wait
  if (!enabled) {
    try {
      log.debug("Disabled, waiting for notification");
      synchronized (lock) {
        lock.wait();
      }
    } catch (InterruptedException e) {
      log.debug("InterruptedException");
    }
  }

  Condition checked without lock held!
  Notification could occur here!
  Could wait forever
Malicious code vulnerabilities
Mutable static

- Can static fields (or the objects they refer to) be modified by untrusted code?
  - Public, non-final static fields
  - Public static fields pointing to an array
- Warnings: 254 in rt.jar 1.5-b59, 967 in eclipse-3.0
Mutable static example

- J2SE 1.5 build 63 (released version),
  javax.swing.plaf.metal.MetalSliderUI

  protected static Color thumbColor;
  protected static Color highlightColor;
  protected static Color darkShadowColor;
  protected static int trackWidth;
  protected static int tickLength;
  protected static Icon horizThumbIcon;
  protected static Icon vertThumbIcon;
Returning a reference to an internal array

- Method returns a reference to an array which is still part of the internal representation of the class
- Caller can
  - see changes to the array
  - make their own changes to the array
- Warnings: 407 in rt.jar 1.5-b59, 755 in eclipse-3.0
Who would do such a thing?

- J2SE 1.4.1, java.util.jar.JarEntry
  
  ```java
  public class JarEntry extends ZipEntry {
      Certificate[] certs;
      public Certificate[] getCertificates() {
          return certs;
      }
  }
  ```

- This is the exact same design flaw as the JDK1.1 code signing flaw
  - That flaw was easily exploitable
Introduction

Bug Patterns

Evaluation

Conclusions
Accuracy goal

- Our goal is that at least 50% of high and medium priority warnings should be real bugs
  - We manually classified warnings produced by tool for several real applications and libraries
- In general, we came close to achieving our goal
  - Some detectors more accurate than others
  - Detectors work better on some applications than others
    - Different development teams use different idioms
- Lower accuracy is tolerable if detector produces a small number of warnings, and real instances are especially serious
Detectors that produce false positives

- For high and medium priority warnings, selected detectors:

<table>
<thead>
<tr>
<th>Application</th>
<th>Warnings</th>
<th>Serious</th>
</tr>
</thead>
<tbody>
<tr>
<td>classpath-0.08</td>
<td>93</td>
<td>66%</td>
</tr>
<tr>
<td>rt.jar 1.5 build 59</td>
<td>349</td>
<td>68%</td>
</tr>
<tr>
<td>eclipse-3.0.0</td>
<td>420</td>
<td>65%</td>
</tr>
<tr>
<td>drjava-stable-20040326</td>
<td>13</td>
<td>77%</td>
</tr>
<tr>
<td>jboss-4.0.0RC1</td>
<td>118</td>
<td>47%</td>
</tr>
<tr>
<td>jedit-4.2pre15</td>
<td>22</td>
<td>50%</td>
</tr>
</tbody>
</table>
These warnings may not be relevant for every application

<table>
<thead>
<tr>
<th>Application</th>
<th>Eq</th>
<th>HE</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>classpath-0.08</td>
<td>2</td>
<td>14</td>
<td>39</td>
</tr>
<tr>
<td>rt.jar 1.5.0 build 59</td>
<td>9</td>
<td>55</td>
<td>259</td>
</tr>
<tr>
<td>eclipse-3.0</td>
<td>3</td>
<td>170</td>
<td>1,000</td>
</tr>
<tr>
<td>drjava-stable-20040326</td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>jboss-4.0.0RC1</td>
<td>1</td>
<td>18</td>
<td>227</td>
</tr>
<tr>
<td>jedit-4.2pre15</td>
<td></td>
<td>6</td>
<td>53</td>
</tr>
</tbody>
</table>
Total warnings counts for FindBugs and PMD
- Using recommended rules for PMD

<table>
<thead>
<tr>
<th>Application</th>
<th>KLOC</th>
<th>FindBugs</th>
<th>PMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt.jar 1.5.0 build 59</td>
<td>1,183</td>
<td>3,314</td>
<td>17,133</td>
</tr>
<tr>
<td>eclipse-3.0</td>
<td>2,237</td>
<td>4,227</td>
<td>25,227</td>
</tr>
</tbody>
</table>

- Style checkers tend to produce a large number of warnings
  - They are most useful to *enforce* consistent standards
- Bug checkers can be used productively on any software
Why do bugs occur?

- Everybody makes dumb mistakes
  - *Everybody*
- Java (and similar languages) have very large standard libraries
  - Many possibilities for confusion and misuse
- Many possibilities for latent bugs (e.g., hashcode/equals)
- Programmers play fast and loose with threads
All software contains bugs
- Including your software
- Some of them are blatant, obvious, and undiscovered
- Very simple techniques suffice to find them

Imprecise analysis can be very accurate

Writing bug detectors is surprisingly easy
- You should try it!
- FindBugs is open source
- We welcome contributions
Future work

- Find, implement detectors for new bug patterns
- Better integration into continuous development
  - False warning suppression
  - Automatic ranking of new warnings based on previously classified warnings
- Bug patterns for beginning programmers
- User-specified patterns
  - Learn new patterns from examples?
Questions?