Code analysis, quality and security overview

Christoph
July 26th 2017
• PhD on reflective OS architectures
• FOSS enthusiast (Linux fan since kernel 0.95)
• Tech support @ FraLUG (including making the coffee)
• IT Sec interests include:
  – Social engineering
  – Cognitive and behavioural psychology
  – SDLC process optimizations and S/W quality
  – Other assorted forms of witchcraft 😊
1. Scope
2. ISO 9126 metrics
3. Attack surface analysis
4. Other observations
Executive summary

- **What is it?**
  - Android runtime environment (similar to libc in standard Linux systems)
  - Glue between kernel and remaining application stack (including Java VMs)

- **Why is it important?**
  - Basis for all applications – any security issues impact other userland

- **What implication does this have?**
  - Attack surface analysis
  - And mitigation
• Assess Bionic code base:
  – Against ISO 9126 maintainability aspects
  – Identify high-level attack surface
  – Additional findings based on further analysis
  – Provide high-level mitigation advice
• **Tools:**
  – SonarQube
  – RATS (Rough Auditing Tool for Security)
  – Cppcheck
  – Common sense and more than 30 years of software development expertise
  – Various other forms of dark magic 😊

• **Codebase:**
  – As found on android.googlesource.com/platform/bionic.git
• Overall code quality:

• But:
  • Some security risks due to insecure coding practices
  • Also many code smells
  • Extensive use of legacy code
ISO 9126

- International standard for s/w quality evaluation

Functionality

Reliability

Maintainability

Efficiency

Usability

Portability

• Analysability
• Changeability
• Stability
• Testability
**Attributes of Maintainability**

Maintainability =

- *Analysability:* Easy to understand where and how to modify?
- *Changeability:* Easy to perform modification?
- *Stability:* Easy to keep coherent when modifying?
- *Testability:* Easy to validate after modification?
<table>
<thead>
<tr>
<th></th>
<th>Volume</th>
<th>Duplicatio</th>
<th>Unit complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysability</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Changeability</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Stability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
• Software Productivity:
  – xLOC
  – Function points (FPs)
  – ...

• Challenge:
  – Expressiveness of different programming languages
  – Approach: weigh xLOC with industry-standard productivity factor
    ➢ Programming Languages Table
### Programming Languages Table:

<table>
<thead>
<tr>
<th>Language</th>
<th>Level</th>
<th>Avg. # of LOC per FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>Smalltalk/V</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>Objective C</td>
<td>12</td>
<td>27</td>
</tr>
<tr>
<td>Haskell</td>
<td>8.5</td>
<td>38</td>
</tr>
<tr>
<td>C++</td>
<td>6</td>
<td>53</td>
</tr>
<tr>
<td>Basic</td>
<td>3</td>
<td>107</td>
</tr>
<tr>
<td>C</td>
<td>2.5</td>
<td>128</td>
</tr>
<tr>
<td>Macro assembler</td>
<td>1.5</td>
<td>213</td>
</tr>
</tbody>
</table>
• Why this matters:
  – Total cost
  – Effort to rebuild overall code base

• Bionic volume metrics:

<table>
<thead>
<tr>
<th>Unit</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total LOCs</td>
<td>422,969</td>
</tr>
<tr>
<td>Files</td>
<td>3,981</td>
</tr>
<tr>
<td>Functions</td>
<td>5,597</td>
</tr>
<tr>
<td>Classes</td>
<td>9,336</td>
</tr>
<tr>
<td>Statements</td>
<td>63,664</td>
</tr>
</tbody>
</table>
• Duplication of code reduces maintainability
  – Substantial duplication implies high maintenance costs
  – Substantial duplication makes bug fixing harder
  – Substantial duplication makes testing harder
• Bionic duplication metrics:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>0.9%</td>
</tr>
<tr>
<td>Blocks</td>
<td>127</td>
</tr>
<tr>
<td>Files</td>
<td>49</td>
</tr>
</tbody>
</table>
• Unit complexity is measured by McCabe’s Cyclomatic Complexity
  – Number of decision points (DPs) per unit (method/function/file)
  – McCabe, IEEE Transactions on Software Engineering, 1976
  – Higher complexity makes units harder to test and change

• For C/C++/Objective C, increment DPs for:
  function definitions, while, do while, for, throw statements, return (except if it is the last statement of a function), switch, case, default, &&, ||, ?, catch, break, continue, goto
**Overview:**

<table>
<thead>
<tr>
<th>Cyclomatic complexity</th>
<th>Risk estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 10</td>
<td>Clear code, small risk</td>
</tr>
<tr>
<td>11 - 20</td>
<td>Complex, medium risk</td>
</tr>
<tr>
<td>21 - 50</td>
<td>Very complex, high risk</td>
</tr>
<tr>
<td>&gt; 50</td>
<td>Not understandable, testability issues, very high risk</td>
</tr>
</tbody>
</table>
• Bionic complexity metrics:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>3.4</td>
</tr>
<tr>
<td>Class</td>
<td>0.2</td>
</tr>
<tr>
<td>File</td>
<td>5.7</td>
</tr>
</tbody>
</table>
• Code analysis result: very good
  – SQALE rating: A
  – Est. technical debt: 17d

• But some security issues:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Occurences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerabilities</td>
<td>1</td>
</tr>
<tr>
<td>Minor issues</td>
<td>74</td>
</tr>
<tr>
<td>Smells</td>
<td>1,634</td>
</tr>
</tbody>
</table>

(SQALE: Software Quality Assessment based on Lifecycle Expectations)
• Good news:
  – No major refactoring required

• Attack surface analysis:
  – Only one major vulnerability
  – Minor issues:
    • Time of check / time of use issues
    • Potential memory leaks
    • Class initialization omissions
• Attack surface analysis (ctd.):
  – Smells: mostly string and buffer handling issues
  – Primarily due to extensive reuse of legacy code

• Remedies:
  – Extended code review
  – Deploy static code analysis tools
  – Fix coding issues
linker.cpp (#351): CWE-562, return of stack variable address

```cpp
static bool realpath_fd(int fd, std::string* realpath) {
    std::vector<char> buf(PATH_MAX), proc_self_fd(PATH_MAX);
    __libc_format_buffer(&proc_self_fd[0], proc_self_fd.size(),
                          "/proc/self/fd/%d", fd);

    if (readlink(&proc_self_fd[0], &buf[0], buf.size()) == -1) {
        PRINT("readlink("\"%s\") failed: %s [fd=%d]", &proc_self_fd[0],
               strerror(errno), fd);
        return false;
    }

    *realpath = &buf[0];
    return true;
}
```
• Typical smells:
  
  – libc/arch-mips/string/memcpy.c: no check on len
    ```c
    memcpy (void *a, const void *b, size_t len) __overloadable
    ```
  
  – libc/arch-arm/bionic/atexit_legacy.c: non-constant format string
    ```c
    static char const warning[] = "WARNING: generic atexit() called from legacy shared library\n";
    __libc_format_log(ANDROID_LOG_WARN, "libc", warning);
    fprintf(stderr, warning);
    ```
1. Reduce attack surface by eliminating security risks (cf. previous slide)
2. Reduce complexity of selected modules
3. Reduce minor duplication by restructuring selected code base portions
4. Identify large volume units and restructure code base as applicable
• Sound code base despite legacy character
• Minimal attack surface requires no major refactoring
• Minor issues can be addressed without much effort
➢ Robust code base for remaining userland
Software sources

• Bionic source code: android.googlesource.com/platform/bionic.git
• Sonarqube: www.sonarqube.org/downloads
• Cppcheck: cppcheck.sourceforge.net
• RATS: code.google.com/archive/p/rough-auditing-tool-for-security/downloads
Discussion / questions
Thank you!

© 2017 CC BY-SA

Christoph
monochromec@gmail.com