BIONIC

Code analysis, quality and security overview

Christoph

July 26th 2017

stat /proc/self

OWASP
The Open Web Application Security Project

- 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 ☺

Overview

1. Scope
2. ISO 9126 metrics
3. Attack surface analysis
4. Other observations

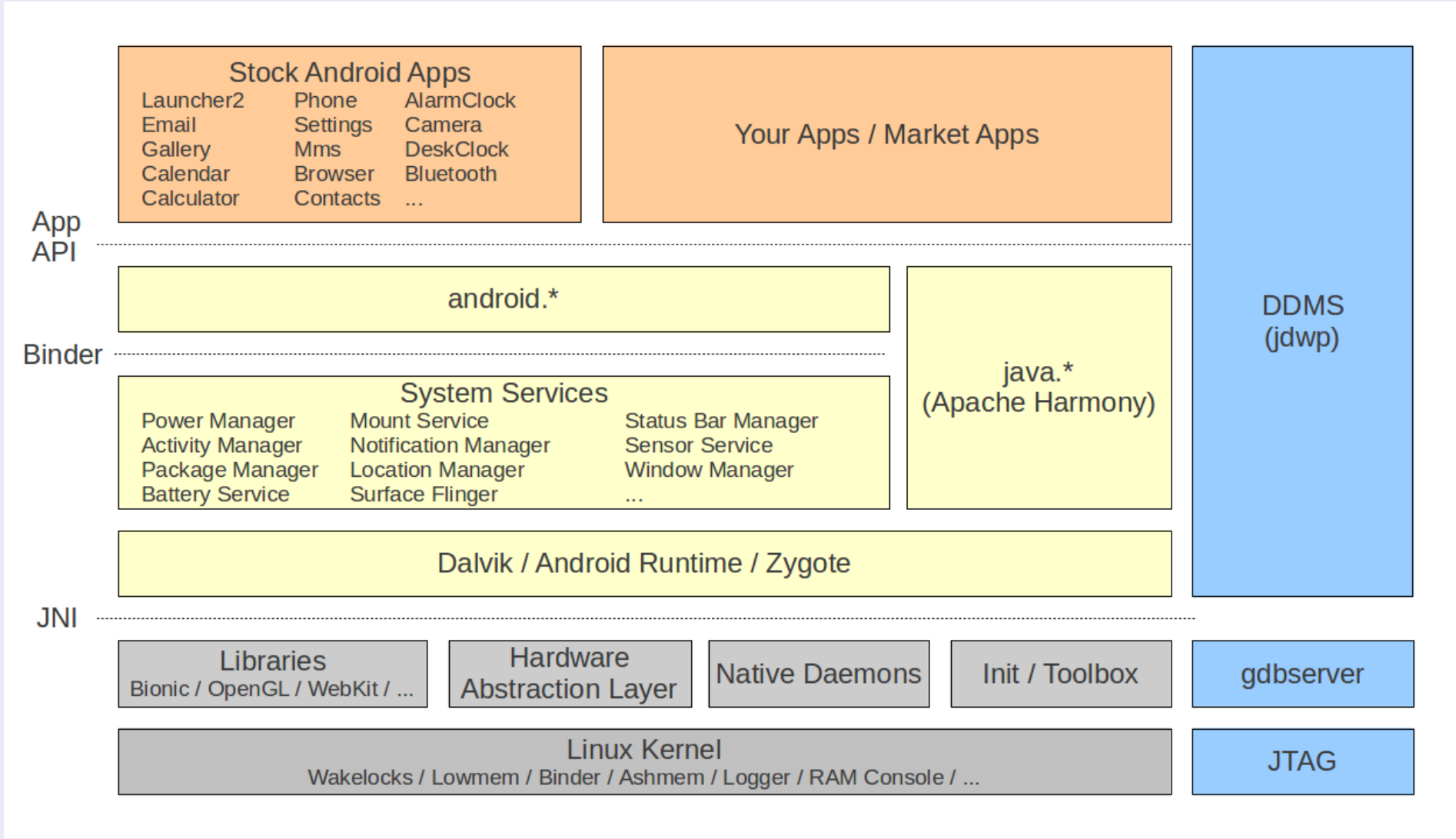
Executive summary

OWASP
The Open Web Application Security Project

- 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

Android overview

OWASP
The Open Web Application Security Project



Scope

- 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

Assessment basis

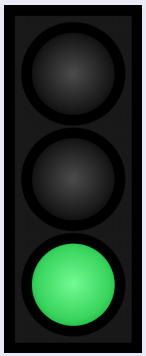
OWASP
The Open Web Application Security Project

- 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

Summary

OWASP
The Open Web Application Security Project

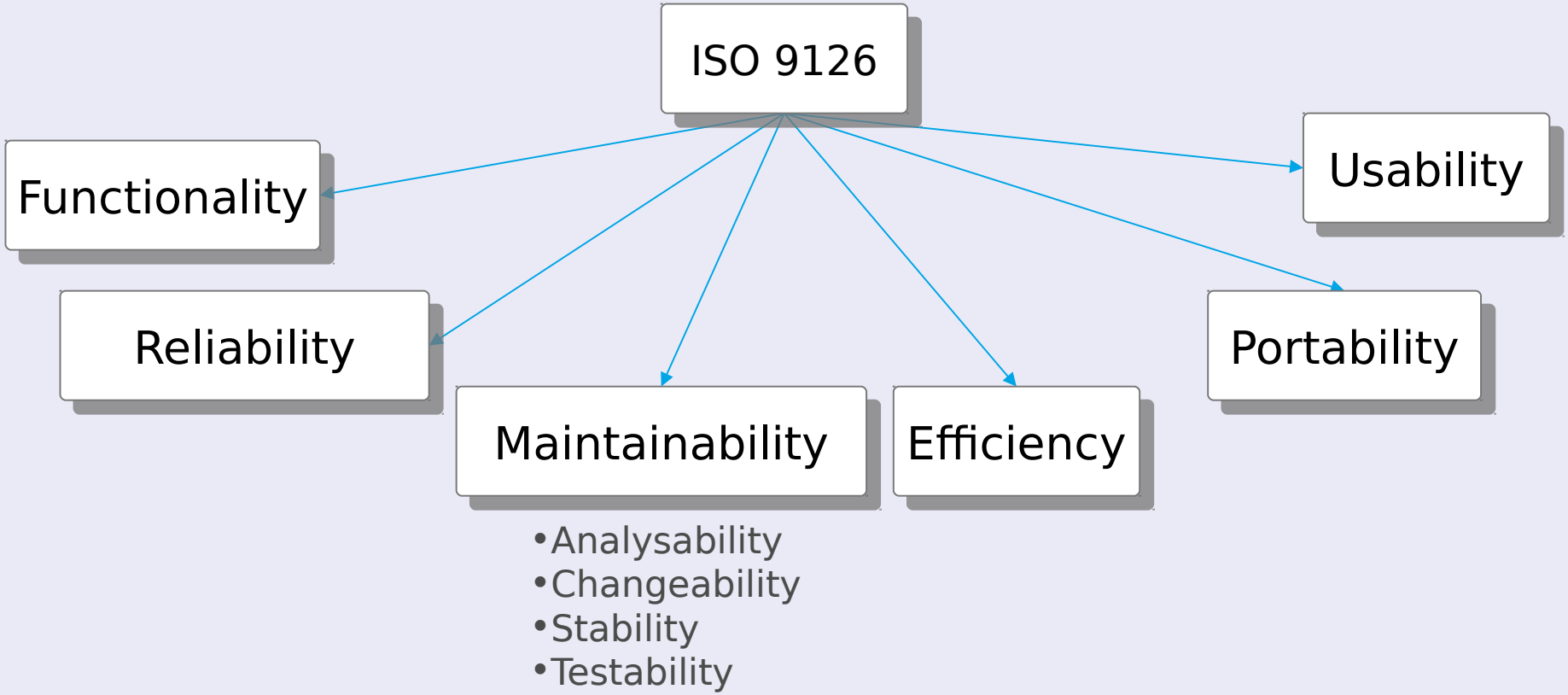
- Overall code quality:



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

ISO 9126

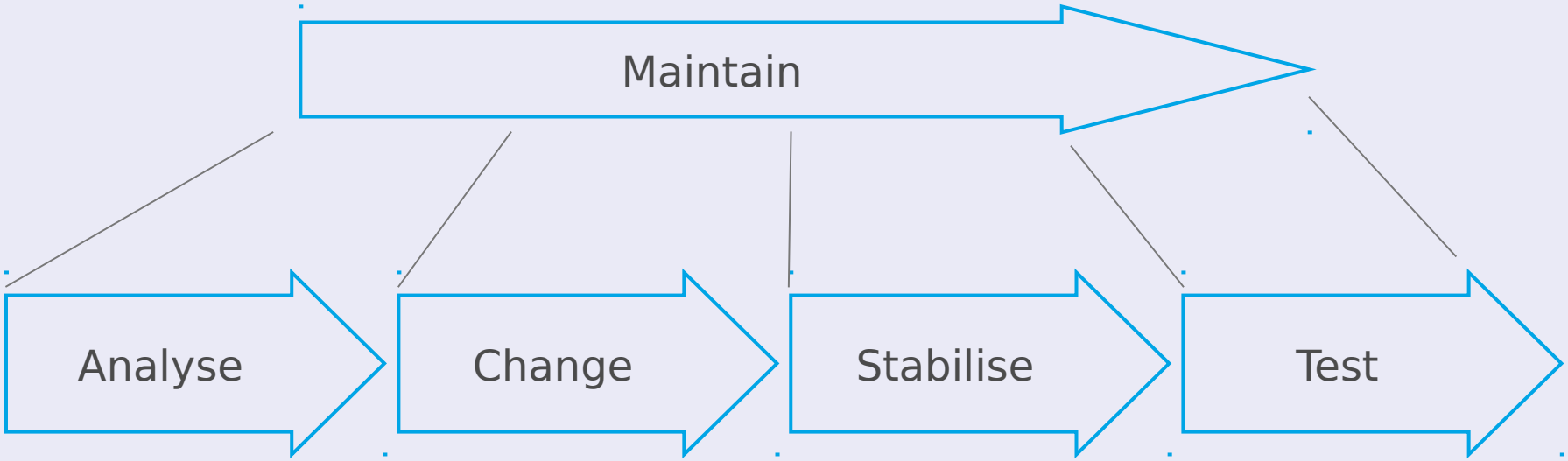
- Internatonal standard for s/w quality evaluation



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?



Simplified assessment model

OWASP
The Open Web Application Security Project

	Volume	Duplication	Unit complexity
Analysability	X	X	
Changeability		X	X
Stability			
Testability			X

Volume

- Software Productivity:
  - xLOC
  - Function points (FPs)
  - …
- Challenge:
  - Expressiveness of different programming languages
  - Approach: weigh xLOC with industry-standard productivity factor
    - Programming Languages Table

Volume (ctd.)

- Programming Languages Table:

Language	Level	Avg. # of LOC per FP
Perl	15	21
Smalltalk/V	15	21
Objective C	12	27
Haskell	8.5	38
C++	6	53
Basic	3	107
C	2.5	128
Macro assembler	1.5	213

Volume (ctd.)

OWASP
The Open Web Application Security Project

- Why this matters:
  - Total cost
  - Effort to rebuild overall code base
- Bionic volume metrics:

Unit	#
Total LOCs	422,969
Files	3,981
Functions	5,597
Classes	9,336
Statements	63,664

Duplication

OWASP
The Open Web Application Security Project

- Duplication of code reduces maintainability
  - Substantial duplication implies high maintenance costs
  - Substantial duplication makes bug fixing harder
  - Substantial duplication makes testing harder

Duplication (ctd.)

OWASP
The Open Web Application Security Project

- Bionic duplication metrics:

Unit	Duplication
Total	0.9%
Blocks	127
Files	49

Unit complexity

OWASP
The Open Web Application Security Project

- 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

Unit complexity (ctd.)

OWASP
The Open Web Application Security Project

- Overview:

Cyclomatic complexity	Risk estimation
1 - 10	Clear code, small risk
11 - 20	Complex, medium risk
21 - 50	Very complex, high risk
> 50	Not understandable, testability issues, very high risk

Unit complexity (ctd.)

OWASP
The Open Web Application Security Project

- Bionic complexity metrics:

Unit	Complexity
Function	3.4
Class	0.2
File	5.7

Conclusion

OWASP
The Open Web Application Security Project

- Code analysis result: very good
  - SQALE rating: A
  - Est. technical debt: 17d
- But some security issues:

Unit	Occurences
Vulnerabilities	1
Minor issues	74
Smells	1,634

(SQALE: Software Quality Assessment based on Lifecycle Expectations)

Attack surface analysis

OWASP
The Open Web Application Security Project

- 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.)

OWASP
The Open Web Application Security Project

- 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

Attack surface analysis (ctd.)

OWASP
The Open Web Application Security Project

- linker.cpp (#351): CWE-562, return of stack variable address

```
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;
}
```

Attack surface analysis (ctd.)

OWASP
The Open Web Application Security Project

- Typical smells:
  - libc/arch-mips/string/memcpy.c: no check on len

```
memcpy (void *a, const void *b, size_t len) __overloadable
```

  - libc/arch-arm/bionic/atexit_legacy.c: non-constant format string

```
static char const warning[] = "WARNING: generic atexit() called
from legacy shared library\n";
   __libc_format_log(ANDROID_LOG_WARN, "libc", warning);
    fprintf(stderr, warning);
```

Overall remedies

OWASP
The Open Web Application Security Project

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

Conclusion

OWASP
The Open Web Application Security Project

- 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

OWASP
The Open Web Application Security Project

- 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

OWASP
The Open Web Application Security Project

Discussion / questions

OWASP
The Open Web Application Security Project

Thank you!

Christoph

monochromec@gmail.com