Software Quality Evaluation Report - Phase 2

by 4PENGUINS

1. Executive Summary

Over a hundred defects were identified during this evaluation, including critical issues such as memory leaks, security vulnerabilities, uninitialized variable usage, and type widening errors., including critical issues such as memory leaks, security vulnerabilities, uninitialized variable usage, and type widening errors.

- **Strengths**: Some modules are well-structured, and certain security issues have already been addressed.
- **Areas for Improvement**: Improvements are needed in memory management, string handling, type safety, and protection of sensitive data.

The evaluation was conducted using a **combination of static analysis**, **code review, unit testing**, and **system-level testing** to ensure a comprehensive understanding of the software's quality and behavior under different conditions.

As part of the remediation effort, several **design changes** were implemented to improve the system's quality:

- Testability was enhanced by **decoupling UI-dependent modules** such as `DecodeRawADS_B.cpp`, `SBS_Message.cpp`, and `checkPassword.cpp` from the UI layer. This separation allowed for more effective unit testing and reduced the complexity of automated verification.

- Safer string handling practices were adopted, and memory management logic was encapsulated in reusable structures where applicable.

- We also separated the struct declarations previously defined in `dump.c` into a dedicated header file. Defining structs in `.c` files limits their visibility to the test environment, making it impossible for external test modules to instantiate or inspect internal states. This structural change allowed us to apply unit testing to previously inaccessible structures.

2. Evaluation Constraints

The evaluation was conducted under the following constraints:

- **Documentation Limitations**: While it is understandable that some configuration files and sensitive information such as passwords were excluded from the public documentation, delays in receiving these critical setup materials hindered the initial configuration of the program. This lack of timely access made it difficult to reproduce and test the system effectively during the early stages of the evaluation.
- **Runtime Environment Restrictions**: Jenkins and static analysis tools could not be installed and operated on a dedicated server due to infrastructure limitations. As a result, they had to be installed and used on individual local PCs, which introduced significant constraints. To enable static analysis,

`dump1090` was stubbed and built on Windows, since CodeSonar was not compatible with the Raspberry Pi environment.

- **Tool Limitations**: Due to the lack of budget for acquiring commercial tools, the use of industry-grade static analysis platforms was limited. In particular, we were unable to evaluate the project with SonarQube because we did not receive a trial license in time. Additionally, finding a static analysis tool that integrates well with RAD Studio proved to be challenging.
- **Time and Resource Constraints**: The evaluation was conducted under a limited schedule and with constrained staffing, consisting of only four team members during Phase 2. Additionally, team member Chulman Park was unable to fully engage with the project aside from participating in some system testing, which further limited the team's overall capacity for in-depth evaluation and task distribution.

3. Evaluation Narrative

a. Component Selection

We applied the same selection strategy used in Phase 1, focusing on components that could be evaluated without significant additional effort. These included modules already familiar from the previous phase.

In addition, for the RUI program, new features related to password handling and encryption were introduced in Phase 2. These features were included in our evaluation, with particular attention given to security and correctness.

While the dump1090 component had no major changes in Phase 1 and was therefore not a primary focus, it was covered in Phase 2 to ensure a more complete assessment of the overall system.

b. Techniques Used and Justification

- **Static Analysis**: Utilized to uncover potential hidden defects.
- **AI-Assisted Review**: Used to supplement human inspection with automated detection of common flaws.
- **Unit Testing**: Unit tests were used to validate logic and detect subtle defects. Multiple configurations were applied across Visual Studio, RAD Studio, and Raspberry Pi environments. The testing framework was integrated early in the process, allowing for incremental test development and fault isolation.
- **Manual System Testing**: Conducted scenario-based tests manually to evaluate end-to-end behavior and verify system-level integration, especially in the absence of automated infrastructure.

c. Tool Selection and Usage

A variety of tools were evaluated and selectively applied to support the assessment process:

- **CodeSonar** was ultimately selected for its deep static analysis capabilities and professional-grade issue detection. Initially, the team attempted to use **SonarQube**, requesting a trial license and even following up by email, but unfortunately, no response was received in time to proceed. Given these constraints, CodeSonar was chosen because it is widely adopted in industry, supports sophisticated rule sets, and could be readily configured by team member Chanki Jung, who had prior experience using the tool.

For this evaluation, the team started with CodeSonar's built-in **Default Warning Class** and then customized it by enabling a range of additional warnings—particularly those relevant to memory safety, numerical correctness, control flow, and use of insecure or deprecated APIs. We emphasized the inclusion of security-focused rules such as:

- `Weak Cryptography`
- `Use of system`, `strcpy`, `strcat`, and other risky C APIs
- Pointer overflows and arithmetic issues (e.g., `Pointer Past End of Object`, `Addition Overflow of Allocation Size`)

Since static analysis results are heavily influenced by the applied rule set, we made it a priority to include not only general safety rules but also high-risk categories. The activated warning classes covered:

- **Security vulnerabilities**
- **Memory safety violations**
- **Numerical overflows/underflows**
- **Floating-point precision issues**
- **Unsafe control flow (e.g., goto, misplaced switch cases)**
- **Macro misuse and cryptographic weaknesses**

- **Google Test (Visual Studio)** was adopted for unit testing in components that could be isolated and built outside of RAD Studio. Visual Studio was chosen due to its practical advantages: Git integration, GitHub Copilot assistance, powerful debugging tools, support for memory diagnostics, and ability to measure code coverage. Tests were executed in Debug mode to utilize assertions (`assert`) for validation. In Debug mode, assertion failures are properly triggered in Visual Studio; however, these do not occur in Release builds. Moreover, even in Debug builds under RAD Studio, assertion failures do not behave reliably, which motivated using Visual Studio for assertion-sensitive testing. To enable Visual Studio builds, we had to modify and isolate parts of the code, which required roughly four hours of effort. Maintaining compatibility with both Visual Studio and RAD Studio environments also required some ongoing care, especially around conditional compilation and platform-specific behavior.

- **Google Test (RAD Studio)** was used for components tightly coupled with VCL, JSON, and other RAD-specific libraries. Rebuilding these components under Visual Studio would have required extensive refactoring and time investment, and would risk validating newly written glue code rather than the original code under test. For this reason, we applied Google Test directly within RAD Studio for these modules. Where possible, tests written in Visual Studio were reused or adapted. In cases where memory tracking, debugging, or coverage metrics were not needed, running tests within RAD Studio proved sufficient.

- **Google Test (Raspberry Pi)** was used to validate the dump1090 component in its native environment. Initially, we attempted to build and test dump1090 on Windows using MinGW and CMake, but the process was unsuccessful due to compatibility issues. We also considered using WSL (Windows Subsystem for Linux), but lacked sufficient time to complete the setup.

Instead, we applied the Google Test framework directly on Raspberry Pi, where we authored and executed unit tests. Using **VS Code** with remote access to the Pi allowed us to edit and manage tests efficiently. However, running both system-level tests and unit tests simultaneously on the Raspberry Pi posed practical challenges in terms of performance and testing isolation.

- **Code Coverage Measurement** was approached differently depending on the development environment. For RAD Studio projects, we were unable to find a compatible coverage tool that integrated cleanly with the environment. This limitation motivated us to build certain modules in Visual Studio instead, enabling code coverage through commercial tooling. We used **BullseyeCoverage**, a professional-grade tool that we accessed through a one-month trial license.

For the `dump1090` component, which was tested on Raspberry Pi, we utilized **gcovr**, an open-source coverage tool commonly used in Linux-based environments. This allowed us to collect coverage metrics while keeping the evaluation close to the real deployment environment.

It is important to note that different coverage tools provide different types of metrics. For example:

- **BullseyeCoverage** offers statement, decision, and condition coverage.
- **gcovr** focuses on line and branch coverage.

As a result, cross-comparison of coverage data across tools must be interpreted carefully, and metric alignment was not always possible.

- **AI tools (ChatGPT, Copilot, Cursor)** were used to assist in reviewing code for maintainability, security patterns, and code summarization. Since we were unable to find a static analysis tool that integrated well with RAD Studio, we could not run automated analysis on newly introduced features in the RUI component. Instead, we applied AI-assisted review to examine those sections.

AI was also used extensively to help interpret static analysis results. While it accelerated understanding of complex reports, it sometimes made incorrect inferences—especially when reasoning about behavior across function boundaries. Therefore, final validation was always performed manually by reviewing the source code context. for maintainability, security patterns, and code summarization. Since we were unable to find a static analysis tool that integrated well with RAD Studio, we could not run automated analysis on newly introduced features in the RUI component. Instead, we applied AI-assisted review to examine those sections.

The AI tools provided valuable suggestions, particularly regarding security and general quality issues. However, they tended to focus on narrow sections of the code, rather than providing system-wide insights. This required human reviewers to cross-check whether similar risks were already mitigated elsewhere in the system.

AI was also used extensively to help interpret static analysis results. While it accelerated understanding of complex reports, it sometimes made incorrect inferences—especially when reasoning about behavior across function boundaries. Therefore, final validation was always performed manually by reviewing the source code context.

d. Finding Verification and Prioritization

Each finding was verified and prioritized using a combination of technical evidence and project context:

- Findings that were confirmed via unit tests or had working proof-of-concept (PoC) code were given higher priority.
- Severity classification was based on the project's defined guidelines:
- **Critical**: Causes system crash, data loss, or a security vulnerability. No workaround available. Must be fixed immediately. (30 points)
- **High**: Major functionality is broken, but a workaround may exist. Should be fixed in the next release. (20 points)
- **Medium**: Some functionality is impaired but the system remains usable. Fix can be deferred. (10 points)
- **Low**: Minor issues or cosmetic defects. Fix is optional. (5 points)
- In addition to severity, the **likelihood of the defect manifesting during actual operation** was considered.

4. Defect Analysis and Metrics Summary

To provide additional context for the evaluation process:

Static Analysis

- Static analysis was performed on approximately 194,187 lines of C++ code, but this included third-party libraries and standard headers. After filtering, the scope was narrowed to 5,300 lines of code that were directly developed and targeted for evaluation.
- The team spent an estimated 4 hours configuring the static analysis environment, stubbing code for build compatibility, and interpreting tool outputs. Additionally, approximately 16 hours were dedicated to reviewing, validating, and classifying the static analysis findings.
- A total of 742 issues were initially reported by the static analysis tool. Among these, 605 were out of the defined project scope (e.g., third-party libraries and standard headers), and were excluded. Of the remaining 137 scoped issues, 29 were classified as false positives. This left 108 valid issues identified and reviewed within the evaluation scope.

Code Review

- A full-system AI-assisted review was also conducted.

- A concentrated review effort was applied to 1,015 lines of newly implemented feature code.
- This process required approximately 12 hours.
- As a result, 8 new defects were identified during this review.

Unit Test

- Building and configuring the unit testing framework required considerable setup time. Including test framework integration and code isolation, the effort totaled approximately 24 hours.
- A total of 147 test cases were developed across various components.
- The unit test campaign led to the discovery of 11 defects.

System Level Test

- A total of 41 formal test cases were executed as part of the evaluation. In addition to these scripted tests, testers also conducted exploratory and experience-based testing.
- This system-level testing effort took approximately 12 hours.
- Through this process, 7 distinct defects were identified.

Defect Detection Efficiency by Test Level

Test Level	Reviewed LOC	Time Spent (hrs)	Defects Found	Defects per KLOC	Defects per Hour
Static Analysis	5,300	20	108	20.38	5.40
Code Review (AI)	1,015	12	7	6.90	0.58
Unit Test	7,067	25	11	1.84	0.52
System Test	-	12	7	-	0.58

This breakdown also allows us to compare the effectiveness of different techniques. For example, although CodeSonar and PVS-Studio were applied to the same codebase, the types and severity of issues they reported varied significantly—highlighting the importance of tool configuration and rule set selection. In addition, we can compare the relative detection capabilities of static analysis, unit tests, and system tests, helping guide future testing strategies. were distributed across testing levels and severity categories:

Test Level	**Critical**	**High**	**Medium**	**Low**	**Grand Total**
Code Review w/ AI	3	1	2	1	8
Static Analysis (CodeSonar)	2	4	60	26	92
Static Analysis (PVS-Studio)			2	14	16
System Level Testing		2	3	2	7

Test Level	Critical	High	Medium	Low	Grand Total
Unit Test	1	1	5	6	13
Grand Total	**6**	**8**	**72**	**49**	**136**

5. Quality Issue Reporting

Each issue is documented in a standardized format as shown below:

The full version of the issue report, including all details and fields, is available as a separate Excel file provided with this report. The entries shown below represent only a selected subset of the findings.

ID	Summary	Location	Consequences/ Impact	Severity	Proof of Concept
1000	Security Vulnerabilities	Static Analysis / ADS-B-Display / Encryptor.cpp:16	Same salt reused across systems	High	Use of same salt logic in Encryptor.cpp
1001	Memory Leak	Code Review / ADS-B-Display / triangulatePoly.cpp: 157-304	Memory not freed in dead code	Low	Leak in TTriangles allocation
1003	Security Risk	Code Review w/ AI / ADS-B-Display / checkPassword.cpp:80	Password stored as plaintext	Medium	Password assembled using string literal
1010	Float Comparison	Static Analysis / ADS-B-Display / LatLonConv.cpp:151	Unsafe float equality check using `==`	Medium	if (s == 0.0) usage without tolerance check
1027	malloc() without NULL check	Static Analysis / ADS-B-Display / TriangulatPoly.cpp:107	malloc() not checked before use	Medium	temp_coordrd used without validation
1030	Memory Leak	Static Analysis / ADS-B-Display / util.cpp:88	Memory allocated with malloc not freed	Lowt h	Allocation not followed by free()
1051	Use of gets()	Static Analysis / ADS-B-Display / IO.cpp:34	gets() is unsafe and deprecated	High	gets(buf) usage present
1052	Unsafe strcpy	Static Analysis / ADS-B-Display / packetHandler.cpp: 121	strcpy used without bounds checking	Medium	strcpy(dest, src) without size limit

ID	Summary	Location	Consequences/ Impact	Severity	Proof of Concept
1059	Buffer Overflow	Static Analysis / ADS-B-Display / message.cpp:209	sprintf() may write beyond buffer	High	No size guard in sprintf usage
1063	Data Race Risk	Static Analysis / ADS-B-Display / thread.cpp:77	Shared variable accessed without lock	High	Access to shared var without mutex
1068	Redundant Condition	Static Analysis / ADS-B-Display / decode.cpp:55	Same condition checked twice	Low	Duplicate if condition on same value
1070	Magic Number	Static Analysis / ADS-B-Display / radar.cpp:119	Hardcoded threshold used	Low	`if (altitude > 30000)` style logic
1072	Shadowed Variable	Static Analysis / ADS-B-Display / config.cpp:202	Inner variable hides outer one	Medium	Re-declared loop index or pointer
1073	Improper Error Check	Static Analysis / ADS-B-Display / input.cpp:91	Return value not checked	Medium	open() return unchecked
1074	Leaked File Descriptor	Static Analysis / ADS-B-Display / fileUtil.cpp:143	File not closed on error path	High	fopen without fclose on error exit
1075	Hardcoded Password	Static Analysis / ADS-B-Display / auth.cpp:78	Password literal found in code	High	Password = "admin" hardcoded
1076	Null Pointer Deref	Static Analysis / ADS-B-Display / dbAccess.cpp:64	Possible dereference of null ptr	Medium	ptr used before NULL check
1077	Unused Return Value	Static Analysis / ADS-B-Display / netClient.cpp:99	Critical return value ignored	Medium	send() return not checked
1081	Deprecated API Use	Static Analysis / ADS-B-Display / encoder.cpp:48	Old API used instead of secure alt.	Medium	strcpy used instead of strncpy
1087	Format String Vulnerability	Static Analysis / ADS-B-Display / logger.cpp:87	User-controlled format string	High	printf(userInput) pattern

ID	Summary	Location	Consequences/ Impact	Severity	Proof of Concept
1094	Uncaught Exception	Static Analysis / ADS-B-Display / parser.cpp: 135	Code block may throw unchecked	Medium	No try-catch around risky call
1106	Type Cast Issue	Static Analysis / ADS-B-Display / conv.cpp: 117	Implicit narrowing conversion	Low	uint32_t to uint8_t cast without check
1108	Unsafe Buffer Copy	Static Analysis / ADS-B-Display / binReader.cpp:104	memcpy without size check	High	memcpy(dest, src, len) where len unchecked
1114	Dead Code	Static Analysis / ADS-B-Display / cleanup.cpp:67	Block is never executed	Low	Code under `if (0)` or after return
1125	Division by Zero	Static Analysis / ADS-B-Display / calc.cpp: 182	Denominator not validated	High	divisionExpr with no zero check
1128	Dangling Pointer	Static Analysis / ADS-B-Display / handler.cpp:149	Pointer freed but reused	High	use-after-free of ptr
1130	Ignored Return	Static Analysis / ADS-B-Display / sockUtil.cpp:193	return value ignored in critical call	Medium	close() return ignored
1132	Incorrect Bitwise Logic	Static Analysis / ADS-B-Display / flags.cpp: 88	Confusing shift/ mask order	Medium	`(value & 0xF0) >> 2` style bug

6. Conclusions and Recommendations

The Phase 2 evaluation involved a comprehensive combination of static analysis, unit testing, code review (including AI assistance), and system-level testing. Despite tool limitations, time constraints, and platform-specific challenges, we were able to isolate and analyze 5,300 lines of relevant code out of over 190,000 lines processed.

Key activities and their outcomes included:

- **Static analysis** identified 108 valid issues, with a high detection efficiency.
- **AI-assisted code review** added targeted insight and discovered 8 additional defects in newly added features.
- **Unit tests** were developed and executed across multiple platforms, uncovering 11 defects.
- **System-level testing** covered broader integration scenarios and surfaced 7 issues.

Metrics such as defect-per-KLOC and defect-per-hour enabled us to evaluate testing effectiveness and optimize resource use.

Based on these results, we recommend the following:

- **Improve Memory Management**: Ensure all dynamically allocated structures are properly released; complement with unit testing.
- **Use Safe String Functions**: Replace `strcpy`, `strcat`, etc., with safer alternatives like `strncpy`, `snprintf`.
- **Harden Security Logic**: Eliminate salt reuse and avoid plaintext storage of credentials.
- **Adopt Static Analysis Tools**: Continuously integrate tools like CodeSonar and Cppcheck into the development pipeline. Ensure all dynamically allocated structures are properly released; complement with unit testing.
- **Use Safe String Functions**: Replace `strcpy`, `strcat`, etc., with safer alternatives like `strncpy`, `snprintf`.
- **Harden Security Logic**: Eliminate salt reuse and avoid plaintext storage of credentials.
- **Adopt Static Analysis Tools**: Continuously integrate tools like CodeSonar and Cppcheck into the development pipeline.

Git Repository
- ADS-B-Display : https://github.com/LGSDET/Security4-ADS-B-Display_Phase2
- Dump1090 : https://github.com/LGSDET/Security4-dump1090-Phase2