Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 07, July - 2025

RTOS Partitioning and CWE Compliance: Safeguarding Avionics Software from Critical Failures

Patruni Rajshekhar Rao
Software Engineer
FTD INFOCOM
Bangalore, India

Abstract—In modern avionics systems, **Real-Time Operating System (RTOS) partitioning** plays a crucial role in ensuring **safety, security, and reliability** by isolating critical software components. However, software vulnerabilities classified under **Common Weakness Enumeration (CWE)** can still compromise system integrity if not properly mitigated. This paper explores how **CWE rules intersect with RTOS partitioning**, emphasizing their impact on **flight control, mission-critical data handling, and avionics security**. Key vulnerabilities such as **buffer overruns (CWE-120), race conditions (CWE-362), deadlocks (CWE-833), and integer overflows (CWE-190)** are analyzed in the context of **ARINC 653 and other avionics RTOS architectures**. We discuss the **best practices in inter-partition communication** that can prevent these failures, ensuring compliance with **DO-178C, ARP4754A, and FACE standards**. By integrating **CWE-aware development methodologies**, avionics engineers can enhance **real-time fault tolerance**, minimize **safety risks**, and improve **system resilience against cyber threats** in **partitioned RTOS environments**.

Keywords—Safety, CWE, ARINC 653, DO178B/C

I. INTRODUCTION (HEADING 1)

In avionics systems, RTOS partitioning is a fundamental technique that ensures isolation between critical software components, preventing faults in one partition from compromising the entire system. However, improper implementation of partitioning can introduce vulnerabilities that lead to catastrophic failures. The Common Weakness Enumeration (CWE) framework provides a structured approach to identifying and mitigating software weaknesses that can arise in partitioned RTOS environments.

Safety-critical avionics software must comply with stringent industry standards such as DO-178C and ARINC 653, where robust partitioning guarantees real-time execution, memory protection, and controlled inter-partition communication. However, buffer overflows (CWE-120), race conditions (CWE-362), and deadlocks (CWE-833) can still emerge within or across partitions if not properly addressed. By integrating CWE-compliant development practices, avionics engineers can ensure fault containment, predictable execution, and high reliability in embedded flight software.

This article explores how RTOS partitioning intersects with CWE rules, highlighting key vulnerabilities and mitigation strategies that enhance avionics software resilience.

II. OBJECTIVE

To ensure the safety and reliability of avionics software by implementing RTOS partitioning and CWE compliance, thereby preventing software vulnerabilities such as buffer overruns, race conditions, and deadlocks that could lead to system failures.

Mathematical Expression:

Let:

- S = System reliability
- P = Number of RTOS partitions
- Vi = Probability of a vulnerability in partition iii
- Mi= Mitigation effectiveness for partition iii
- R = Overall risk factor due to software vulnerabilities

The objective is to **minimize risk** and **maximize system reliability**, expressed as:

S=1−R

$$R - \sum_{i-1}^{P} V_i \cdot (1 - M_i)$$

where:

- Vi is reduced by applying CWE rules (e.g., buffer protection, race condition handling).
- Mi is increased by using **memory isolation, time partitioning**, and **deterministic execution** in RTOS.

Thus, the goal is to **maximize Mi** for all partitions to achieve **S→1** (i.e., a highly reliable system with minimal risk).

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 07, July - 2025

III. PROBLEM STATEMENT

In modern safety-critical avionics systems, ensuring the reliable operation of software is paramount. RTOS partitioning is a key approach to isolate different software components into distinct, time-controlled partitions to prevent interference and ensure real-time constraints are met. However, the integration of partitioning with CWE (Common Weakness Enumeration) rules is crucial to avoiding critical vulnerabilities that could lead to system failures or safety hazards.

A significant issue arises when incorrect memory management, uncontrolled access to shared resources, or failure to follow time constraints leads to system instability. These weaknesses, such as buffer overflows, deadlocks, and race conditions, can cause catastrophic effects, especially in real-time operating systems (RTOS), where timing and isolation are essential.

A. Faulty Code in RTOS Partition

Faulty Code in RTOS Partition (Risk of Buffer Overflow)
Imagine an avionics system that collects sensor data and stores it in a buffer. The sensor input is then used for control surface adjustments. In this scenario, if there's no check on the data length, a buffer overflow can occur, leading to memory corruption within the RTOS partition.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BUFFER_SIZE 128

void process_sensor_data(const char *sensor_input) {
   char data_buffer[MAX_BUFFER_SIZE];
   strcpy(data_buffer, sensor_input);   // CWE-120: Buffer Overflow, no bounds checking
   printf("Processed data: %s\n", data_buffer);
}

int main() {
   const char *large_input = "This input string is larger than the buffer and will cause an overflow!";
   process_sensor_data(large_input);   // This call causes a buffer overflow
   return 0;
}
```

B. Situational Analysis

Problem with Partitioning and Buffer Overflow:
In a partitioned RTOS like PikeOS, where the Flight Control System (FCS) is isolated from other subsystems (e.g., navigation, communication), the buffer overflow could overwrite adjacent memory belonging to another partition. This can lead to an inter-partition fault or unintended interaction between partitions, violating the isolation principles that RTOS partitioning is designed to enforce.

C. Methodology and Mitigation

In avionics systems, RTOS partitioning is employed to separate critical and non-critical tasks. The primary goal of partitioning is to ensure that faults or failures in one partition do not affect the others. The RTOS partitions are designed to have dedicated resources (memory, CPU time, etc.) to prevent unintended interactions between tasks.

- Time partitioning ensures that each task or partition receives deterministic CPU time without interruptions from other partitions.
- Space partitioning isolates memory resources to prevent one task from corrupting or interfering with another.
- Example: In PikeOS or RTEMS, partitions are defined with specific memory addresses and time slots for execution.

The RTOS scheduler ensures that critical tasks, such as Flight Control Systems (FCS) or Navigation functions, receive higher priority over non-critical background tasks, such as log management or diagnostic checks.

- Tasks within each partition are scheduled based on priorities defined according to their safety-critical importance.
- Non-blocking, priority-driven scheduling prevents task starvation, guaranteeing critical functions are always prioritized.

D. Impact of Partitioning on CWE Mitigation

RTOS partitioning isolates tasks, ensuring that:

- Memory Corruption (e.g., buffer overrun) within one partition does not affect other partitions.
- Critical Tasks have exclusive access to time-sensitive resources, ensuring their performance is not interrupted by non-critical tasks.

This isolation reduces the attack surface, making it harder for potential vulnerabilities in non-critical partitions to propagate and impact the core avionics functions.

E. Feedback Loop for Continous Improvement

By collecting data on partition failures, CWE rule violations, and runtime anomalies, developers can continuously improve the software architecture and mitigate newly discovered vulnerabilities. This feedback loop leads to:

- Improved Coding Practices: Incorporating lessons learned into future code development.

  Enhanced Fault Tolerance: Adapting partition configurations and task schedules to prevent similar failures from occurring.

IV. MITIGATION STRATEGIES

By collecting data on partition failures, CWE rule violations, and runtime anomalies, developers can continuously improve the software architecture and mitigate newly discovered vulnerabilities. This feedback loop leads to:

- Improved Coding Practices: Incorporating lessons learned into future code development.
- Enhanced Fault Tolerance: Adapting partition configurations and task schedules to prevent similar failures from occurring.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 07, July - 2025

A. Partition Isolation and Boundaries

- RTOS partitioning involves dividing an avionics system into isolated partitions, each running its own application with restricted resources. This prevents the failure of one partition from affecting others, a fundamental principle in ARINC 653 or PikeOS-based systems.
- CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer): Partition isolation ensures that tasks within one partition cannot overrun their allocated memory and corrupt another partition's data.

Formal Verification Approach:

- Model Checking: Formal verification tools like UPPAAL or PRISM can model and verify partition isolation properties. By modeling each partition as a finite-state machine (FSM), it can be mathematically verified that a partition's memory bounds will never be exceeded, preventing buffer overruns.
  - Mathematical Algorithm:

```
For each partition i, define:
MemoryRange_i = [start_i, end_i]

Ensure that for every memory access in partition i:
MemoryAccess_i∈ MemoryRange_i
```

This ensures that memory accesses stay within the bounds allocated to each partition.

- Safety Property:

  - Using temporal logic (CTL or LTL), a property can be verified, such as:

```
AG(MemoryAccess_i >= start_i && MemoryAccess_i <= end_i)
```

This checks that memory access always stays within valid bounds.

B. Task Synchronization and Deadlock Prevention

- In avionics systems, multiple tasks may require access to shared resources. Using semaphore-based synchronization or priority-based task scheduling, critical tasks must be executed without interference from non-critical tasks.
- CWE-833 (Deadlock): Partitioning combined with proper synchronization ensures that tasks don't enter deadlock situations.

Formal Verification Approach:

- Algorithmic Approach: Deadlock prevention algorithms like Banker's Algorithm or Resource Allocation Graphs are used to ensure that tasks never enter a deadlock situation by mathematically analyzing resource allocation patterns.
- Mathematical Algorithm:
- Let T be the set of tasks, R the set of resources, and A[i,j] the allocation matrix where i is the task and j is the resource.
- Ensure there is always a safe sequence of task executions:
- SafeSequence(T, R) = {T1, T2, ..., Tn} such that A[i,j] ≤ Available[j] for all i.
- This ensures that tasks are granted resources in a manner that avoids deadlock.
- Formal Verification:

  Use formal methods to verify that no circular wait conditions exist:

  AG(¬(exists cycle in ResourceAllocationGraph))

  This property ensures that the system will never enter a state where tasks are waiting indefinitely for resources.

C. Preventing Buffer Overrun and Memory Corruption

- CWE-120 (Buffer Overflow): RTOS partitioning restricts each partition's memory access, and buffer overruns are prevented by using bounds-checking techniques like memcpy() with size limits or by using dynamic memory managers that perform overflow checks.

Formal Verification Approach:

- Buffer Bounds Check: Use abstract interpretation or static analysis to ensure that no write operation exceeds the bounds of allocated buffers within a partition.
  - **Mathematical Algorithm:** For a buffer access operation, ensure:

```
For each buffer, Buffer_i with size N_i, and write operation w:
if (w >= 0 && w < N_i)
    Access is valid.
else
    Invalid access.
```

- **Formal Property:** The formal verification model would ensure that for every write to buffer **i**:

```
AG(Write_i >= 0 && Write_i < Size_i)
```

This temporal logic formula ensures that buffer writes are always within the valid range.

D. Secure Partitioning and Authentication

- CWE-287 (Improper Authentication): In avionics, sensitive data between partitions or systems must be securely authenticated. Partition boundaries prevent unauthorized access, and partition communications are secured via public key infrastructure (PKI) or HMAC.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 07, July - 2025

Formal Verification Approach:

- Cryptographic Security Model: Formal methods can be used to verify that all communications across partitions are properly authenticated, ensuring that only authorized partitions can access sensitive data.
    - **Mathematical Algorithm**: For partition communication, use **HMAC** for authentication:

      ```
      HMAC(PartitionKey, Data) = Expected_HMAC
      ```

      If the computed HMAC matches the expected value, the data is authenticated.

- Formal Property:

  The system model can verify that communication always involves proper authentication:

  ```
  AG(AuthenticationCheck(Partition_i) == True)
  ```

E. Conclusion

In safety-critical avionics systems, RTOS partitioning combined with CWE compliance provides a robust framework for fault isolation, vulnerability detection, and system reliability. By partitioning critical tasks and applying rigorous static analysis during the development process, avionics software developers can create more secure, fault-tolerant, and mission-ready systems.

This methodology emphasizes the importance of structured development practices and rigorous testing in preventing software failures that could jeopardize flight safety.

TABLE I. IMPACT AND MITIGATION GLIMPSE

CWE ID	Vulnerability	Impact in Avionics RTOS	Mitigation Strategies
CWE -120	Buffer Overflow	Can cause memory corruption, leading to crashes or execution of malicious code.	Use snprintf() instead of sprintf(), apply boundary checks (memcpy_s() or strncpy()), and enable MPU (Memory Protection Unit) in RTOS to prevent illegal memory access.
CWE -125	Out-of-Bounds Read	Leaks sensitive data or causes unexpected behavior.	Implement strict boundary checks, use safe memory APIs, and enable MMU/MPU protections for each RTOS partition.
CWE -787	Out-of-Bounds Write	May corrupt memory or overwrite critical avionics parameters.	Enforce strict RTOS partitioning with memory boundaries, validate all input sizes, and use compiler flags (-fstack-protector or -D_FORTIFY_SOURCE=2).
CWE -415	Double Free	Can cause undefined behavior or system instability.	Implement reference counting, avoid manual memory management in safety-critical avionics tasks, and use RTOS memory pools instead of dynamic allocation.
CWE -362	Race Conditions	Can lead to data corruption in shared avionics resources.	Use RTOS mutexes, semaphores, or message queues to synchronize tasks and prevent concurrent access.
CWE -476	NULL Pointer Dereference	Can cause a system crash in avionics software.	Validate all pointers before dereferencing, use NULL checks, and enable RTOS exception handling to catch segmentation faults.
CWE -798	Hardcoded Credentials	Leads to security breaches and unauthorized access to avionics systems.	Use secure key storage mechanisms, avoid embedding credentials in firmware, and implement runtime credential provisioning.
CWE -918	Server-Side Request Forgery (SSRF)	Can expose avionics communication channels to cyber threats.	Restrict network access, use firewall rules in RTOS, and validate all remote requests before processing.
CWE -119	Improper Restriction of Operations within Memory Bounds	Can lead to stack overflows or execution of arbitrary code.	Use stack canaries, control flow integrity (CFI), and partitioning enforcement in RTOS.
CWE -502	Deserialization of Untrusted Data	Can lead to remote code execution or avionics system hijacking.	Use trusted serialization libraries, implement whitelisting, and perform signature verification of data.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 07, July - 2025

REFERENCES

[1] S. Duzellier, ‘‘Radiation effects on electronic devices in space,’’ Aerosp. Sci. Technol., vol. 9, no. 1, pp. 93–99, Jan. 2005

[2] B. Todd and S. Uznanski, ‘‘Radiation risks and mitigation in electronic systems,’’ in Proc. CAS-CERN Accel. School, Power Converters, Baden, Switzerland, May 2014, pp

[3] T. Kuwahara, Y. Tomioka, K. Fukuda, N. Sugimura, and Y. Sakamoto, ‘‘Radiation effect mitigation methods for electronic systems,’’ in Proc. IEEE/SICE Int. Symp. Syst. Integr. (SII), Dec. 2012, pp

[4] D. P. Siewiorek and R. S. Swarz, ‘‘Faults and their manifestations,’’ in Reliable Computer Systems, 2nd ed., D. P. Siewiorek and R. S. Swarz, Eds., Boston, MA, USA: Digital Press, 1992, pp

[5] M. Barella, G. Sanca, F. G. Marlasca, W. R. Acevedo, D. Rubi, M. A. G. Inza, P. Levy, and F. Golmar, ‘‘Studying ReRAM devices at low Earth orbits using the LabOSat platform,’’ Radiat. Phys. Chem., vol. 154, pp. 85–90, Jan. 2019