TMS-MSR-001-PUB · PUBLIC SUMMARY · APPROVED FOR PUBLIC RELEASE

Memory Safety Roadmap

Asymmetric Collaborative Counter-Swarm — Capability Negotiation Layer

Approved for public release; distribution unlimited. This is the public summary of TMS-MSR-001, published in accordance with CISA and FBI, **Product Security Bad Practices**. It contains no algorithm internals, no descriptor field semantics and no protocol byte layouts. Published 2 August 2026 by Tejas Mission Systems LLC, Wimberley, Texas.

1. Purpose and the Obligation This Answers

Tejas Mission Systems is developing the Asymmetric Collaborative Counter-Swarm capability, whose core is a Capability Negotiation Layer allowing independently hosted platforms to negotiate and allocate effector capability against a swarm threat. An internal architecture decision of 2 August 2026 allocated that core to C++17. C++ is not a memory-safe language. That decision was accepted only on the condition that eight named mitigations become binding, the eighth of which was the publication of this roadmap. This document discharges that commitment.

The external obligation comes from **Product Security Bad Practices**, issued jointly by the Cybersecurity and Infrastructure Security Agency and the Federal Bureau of Investigation. It is voluntary guidance rather than a regulation, but it is the standard a government evaluator or a prime contractor's supply-chain review will read against. It names two bad practices that apply here.

Named bad practice	Applies to Tejas Mission Systems?
The development of new product lines in a memory-unsafe language where readily available alternative memory-safe languages could be used.	**Yes, in part.** This is a new product line in C++17. Section 3 states why an alternative is not readily available for the core, and where that argument does not hold.
For existing products written in memory-unsafe languages, not having a published memory safety roadmap.	**Yes.** The guidance asks that a roadmap be published by the end of 2025. That date has passed. This roadmap is late against the published date, and is being issued before the first line of covered code is written.

Table 1. The two bad practices at issue. Source: CISA and FBI, Product Security Bad Practices.

The guidance also states what a roadmap must contain: a prioritised approach to eliminating memory safety vulnerabilities in priority code components, a demonstration that the plan leads to a significant and prioritised reduction, and evidence of a reasonable effort to follow it. Sections 4, 6 and 7 answer those three requirements in order.

One honest advantage. No production code exists yet. This roadmap is being written before the exposure is created rather than after, which is the opposite of the situation the guidance was drafted to address. Every commitment in Section 5 is a constraint on code not yet written, not a remediation backlog.

2. Language Baseline and Where the Exposure Sits

The software is organised into eight segments. Language allocation and the resulting exposure are as follows.

Segment	Language	Memory-safety status
A1 Negotiation Core	C++17	**Unsafe.** The whole of the negotiation core.
A2 Host Bearer Adapters	C++17	**Unsafe.** Carries wire traffic.
A3 Host Integration Adapters	C++17	**Unsafe.** Parses host-supplied data.
A4 Mission Function	Python 3.10	Safe at application level. See the note below.
A5 Platform Services	C++17, Python tooling	**Unsafe** in the C++ portion.
A6 Test, Simulation, Conformance	Python 3.10	Safe at application level. Off-target, not shipped.
A7 Federation Plane	C++17	**Unsafe.** Network-facing between platforms.
A8 Security and Assurance	C++17, Python tooling	**Unsafe** in the C++ portion.

Table 2. Language allocation and resulting exposure by segment.

> **Python is not a free pass.** CPython is itself written in C, and native extension modules run outside the interpreter's safety guarantees. Python code is treated here as safe from the classes of defect that matter — buffer overruns, use-after-free, uninitialised reads — while acknowledging that the interpreter and any C extension underneath it are not. That is a third-party dependency question and is governed separately.

3. Why C++ and Not a Memory-Safe Language

The CISA language turns on whether readily available alternative memory-safe languages could be used. For this core the honest answer is qualified, and the qualification is stated precisely.

Candidate	Safe?	Why it was not chosen for the initial core
Rust	Yes	The FACE Technical Standard maps its interface definition language only to C, C++, Ada and Java. Choosing Rust for the core removes the open-standard adapter path, which is the largest addressable-market lever in the architecture. This was the deciding factor.
Ada / SPARK	Yes	The only language that is both FACE-supported and named memory-safe. Rejected for the initial effort on **program risk, not technical merit**: no in-house depth, a six-month feasibility window, and a single-person team. It is reserved for safety-critical units of conformance and that reservation remains live at Milestone M4.
Java	Yes	FACE-supported, but unsuited to an embeddable library that an integrator builds into an existing airborne payload. Not pursued.
C++17	**No**	Chosen. Embeddable as a library, FACE-profileable, and the language the target ecosystem already builds in. The exposure this creates is the reason this document exists.

Table 3. Language candidates against the FACE constraint.

Where the argument does not hold. The FACE constraint binds only the code that must present a FACE-conformant interface. It does not bind the test and simulation segment, which is already Python, and it binds platform services and assurance tooling only weakly. Those are the first places a memory-safe language can be introduced at no cost, and Section 6 puts them there.

> **Throughput is not offered as a justification anywhere in this document.** The bench bearer runs at 921600 baud and the framing protocol caps a frame at 255 bytes. No language in Table 3 is too slow for that. Any argument that C++ was chosen for speed would be false and is not made.

4. Priority Code Components

CISA names the priority categories directly: network-facing code, and code handling sensitive functions such as cryptographic operations. Earlier CISA and NSA material adds parsers and codecs to the same set. Mapped onto the architecture, that language identifies specific components.

4.1 Priority 1 — untrusted input crosses the boundary here

Component	Exposure	CISA category
Transaction Manager	Fragmentation, reassembly, acknowledgement and store-and-forward of frames from a peer that is not trusted. Reassembly logic is the classic overflow site.	Network-facing
Canonical Codec	The decoder. Every byte that enters the core passes through it. Length fields, offsets and variable-width members are all attacker-influenced.	Codec / parser
Cryptographic Services	Key handling and cryptographic policy.	Cryptographic operations
Peer Bearer Adapters	Carriage of the federation plane between independently hosted platforms. Untrusted by construction — the peer interface is symmetric, with no master.	Network-facing
Peer Trust and Authentication	Runs before a peer is authenticated, so it processes input from an unauthenticated source by definition.	Network-facing

Table 4. Priority 1 components. These five receive every mitigation in Section 5 without exception, and are the first candidates for language replacement in Section 6.

4.2 Priority 2 — exposed, but behind a Priority 1 component

- **Host bearer adapters.** They touch the wire first, but hand off to the transaction manager and codec without interpreting payload content.
- **Negotiation state machine.** Consumes decoded input. Protected by the codec if and only if the codec is correct.
- **Transcoders and perception input adapters.** Format conversion at the edge, on host-supplied data.
- **Time, frames and resilient positioning.** Consumes external time and position, including in a GPS-denied condition where the input may be adversarial.

4.3 Priority 3 — no external input path

All remaining components. These carry the general obligations of Section 5 but are not scheduled for language replacement.

5. Binding Commitments

Eight mitigations are binding on the program. Each is stated with the artifact that proves it, because a commitment with no evidence is not a commitment. Commitments 1 through 6 are continuous-integration gates.

#	Commitment	Evidence that it is being followed
1	**Hardened C++17 subset.** No raw owning pointers, no manual new or delete in application code, no C-style casts, no variable-length arrays, no unchecked C string functions.	Written into the coding standard and enforced by the static-analysis configuration in the repository. Violations fail the build.
2	**Bounds-checked views.** All buffer traversal through a span or view type carrying a length. No pointer arithmetic on decoded input.	Analyser rule set plus a repository-wide gate counting raw pointer-arithmetic sites. Target: zero in Priority 1 components.
3	**Automatic initialisation.** All objects initialised at declaration; compiler flags set to trap on uninitialised reads.	Compiler flag set recorded in the build toolchain file and printed in the build log.
4	**Sanitizer builds.** Address and undefined-behaviour sanitizers run on the full test suite in continuous integration on every commit.	Continuous-integration job status. A failing sanitizer run blocks merge.
5	**Continuous fuzzing of the decoder and reassembly path.** Coverage-guided fuzzing seeded from the conformance corpus.	Fuzzing job with a recorded corpus, executions counter, and a crash-triage log. Findings enter the defect register.
6	**Static analysis gate.** Full-project static analysis; no new findings at high severity permitted to merge.	Continuous-integration job status plus a dated findings report retained per build.
7	**No hand-written cryptography.** The cryptographic services component binds a validated module and implements no primitive of its own.	Software bill of materials entry naming the module and its validation status.
8	**This roadmap, published and maintained.** Reviewed at every architecture decision that touches language allocation, and at least annually.	This document and its revision history.

Table 5. The eight binding commitments and their evidence artifacts.

> **None of these is in force yet.** No repository, no continuous-integration pipeline and no coding standard file exists as of 2 August 2026. Commitments 1 through 6 are gates that must exist **in the repository before the first source file of the core is committed**, and that ordering is the single most important line in this document. Standing a gate up after the code is written costs an order of magnitude more and never reaches the same coverage.

6. Prioritised Reduction Plan

The guidance asks for a plan that leads to a significant, prioritised reduction, not a wholesale rewrite. The strategy is boundary-first: replace the language where untrusted input first becomes structured data, and leave the interior in C++ until there is a reason to move it. That is the highest defect-density-per-line region of any protocol implementation, and it is also the smallest and most testable, because it has a narrow, fully specified interface.

Milestone	Content	Gate
M1 Gates before code	Coding standard, analyser rule set, sanitizer CI, static analysis gate and pointer-arithmetic counter all present in the repository.	Before the first core commit.
M2 Fuzzing live	Coverage-guided fuzzing running against the decoder and the reassembly path, seeded from the conformance corpus.	Before the first end-to-end negotiation on the bench.
M3 Exposure measured	Line counts and pointer-arithmetic site counts published per component. Establishes the Section 7 baseline that later reduction is measured against.	At the feasibility gate.
M4 Safe-language evaluation	Written evaluation of Ada or SPARK for the codec and cryptographic services specifically, including whether a FACE-conformant unit of conformance can be built in it and how it links against a C++17 core.	Close of the feasibility effort.
M5 First replacement	The canonical codec reimplemented in a memory-safe language behind an unchanged interface, or a written record of why it was not. It is chosen first because it is the narrowest interface in Priority 1 and the highest exposure.	First year of the development effort.
M6 Priority 1 clear	All Priority 1 components in a memory-safe language, or under a formal exception approved and recorded as a numbered architecture decision.	Close of the development effort.

Table 6. Reduction milestones. M1 through M3 are unconditional. M4 through M6 depend on continued funding and are stated as intent, not as a funded commitment.

New code goes in a safe language wherever the FACE constraint does not reach. The test and simulation segment is already Python. Platform-services tooling and assurance generators have no conformance requirement and will not be written in C++ unless a specific reason is recorded. This costs nothing and it steadily moves the ratio in Section 7.

7. How Reduction Is Measured

Five measures, all machine-generated from the build, all published with each milestone. Judgement is deliberately excluded.

Measure	Definition	Target
Unsafe-language ratio	Source lines in memory-unsafe languages as a fraction of total first-party source lines, reported per priority tier.	Falling at every milestone. No absolute target set in the feasibility phase.
Priority 1 parser coverage	Fraction of Priority 1 decode and reassembly lines in a memory-safe language.	Zero at M1. Non-zero at M5. 100 percent or a recorded exception at M6.
Pointer-arithmetic sites	Count of raw pointer-arithmetic expressions in Priority 1 components, from the static analysis pass.	**Zero**, from M1 onward.
Sanitizer status	Consecutive clean sanitizer runs of the full suite.	Clean on every commit. One failure blocks merge.
Fuzzing exposure	Cumulative fuzz executions and unique crashes found and closed on the decoder and reassembly path.	Monotonically rising executions; zero open crashes at any gate.

Table 7. Roadmap metrics. Each is produced by the build system and requires no manual assessment.

8. Ownership, Review and Contact

- **Owner.** G. Anthony Carpio-Peña, Principal Investigator, Tejas Mission Systems LLC.
- **Contact.** anthony.pena@tejasmissionsystems.com
- **Review triggers.** Any architecture decision that changes language allocation; any new segment or Priority 1 component; each milestone in Table 6; and annually in any case.
- **Exceptions.** A Priority 1 component may remain in a memory-unsafe language only under an exception recorded as a numbered architecture decision stating the reason, the compensating controls and the review date. No exception is granted by silence.
- **Relationship to the internal document.** This is the public summary of TMS-MSR-001. The internal document adds scope boundaries, component-level identifiers and cross-references to the controlled architecture baseline. Nothing in the internal document contradicts anything stated here.

9. References

CISA and FBI, **Product Security Bad Practices**.
https://www.cisa.gov/resources-tools/resources/product-security-bad-practices

CISA and FBI, **Product Security Bad Practices** (advisory copy).
https://www.ic3.gov/CSA/2024/241016-2.pdf

CISA and NSA, **Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development**, June 2025.
https://media.defense.gov/2025/Jun/23/2003742198/-1/-1/0/CSI_MEMORY_SAFE_LANGUAGES_REDUCING_VULNERABILITIES_IN_MODERN_SOFTWARE_DEVELOPMENT.PDF

CISA Cybersecurity Advisory Committee, **Recommendations on Memory Safety**, December 2023.
https://www.cisa.gov/sites/default/files/2023-12/CSAC_TAC_Recommendations-Memory-Safety_Final_20231205_508.pdf

Office of the National Cyber Director, **Back to the Building Blocks: A Path Toward Secure and Measurable Software**, February 2024.
https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

The Open Group, **FACE Technical Standard overview**.
https://www.opengroup.org/sites/default/files/TWG_Overview_2021_TIM.pdf

AdaCore, **AdaCore Technologies for FACE**.
https://www.adacore.com/uploads/books/AdaCoreTechnologiesForFace-V1_02_final.pdf