Memory Safety Checklist:

Secure Your C/C++ Codebases

UNDERSTAND THE RISKS

- Buffer Overflows
- Use-After-Free Errors
- Dangling Pointers
- Memory Leaks

USE MEMORY-SAFE COMPILER OPTIONS

- Enable stack canaries
- Use address sanitizers
- Enable bounds checking and control-flow protections

ADOPT SAFE CODING PRACTICES

- Avoid unsafe functions
- Validate all inputs and array bounds
- Conduct manual code reviews
- Use trusted libraries only

APPLY STATIC AND DYNAMIC ANALYSIS TOOLS

- Use static analyzers
- Run dynamic analysis tools
- Regularly scan for memory leaks, buffer overflows, and use-after-free bugs

SECURE NEW AND LEGACY CODE

(WITHOUT CODE REWRITES)

- **Apply Runtime Protections**
  - Control Flow Integrity (CFI)
  - Load-time Function Randomization (LFR)
- **Patch Regularly**
  - Prioritize by CVSS scores

AUTOMATE SECURITY TESTING IN CI/CD

- Embed fuzz testing in your pipelines
- Run regression and memory safety tests with every commit
- Monitor for new CVEs relevant to your dependencies

USE MEMORY SAFE LIBRARIES WHERE POSSIBLE

- Swap out risky modules for memory safe alternatives (Rust FFI, hardened libc)
- Adopt well-maintained, vetted open-source components

RESPOND TO VULNERABILITIES QUICKLY

- Patch fast, especially for known memory corruption flaws
- Use runtime protection tools to mitigate exploit windows

PLAN FOR THE FUTURE

- **Selective Rewriting in Rust or Go**
  - Focus on critical components
- **Write a Memory Safety Roadmap**
  - Follow CISA's Secure by Design guidance

BE PROACTIVE ABOUT MEMORY SAFETY

Reduce risk in embedded devices and critical systems written in C/C++. Explore tools like RunSafe Protect to harden code against memory safety vulnerabilities with no rewrites required.

LEARN MORE