Core Infrastructure Initiative (CII) Badging

The Best Practices Program is an open source secure development maturity model. Projects having a CII badge will showcase the project's commitment to security. Open source project maintainers answer a short questionnaire to be awarded a "Best Practices Badge".

In order for an O-RAN-SC project to be awarded the basic "passing" grade, a project should score 100% on the following questions. If a question is not applicable, select N/A.

Project	Entered in wiki	CII Badge Status
RIC Applications	Y	blocked URL
Near Realtime RAN Intelligent Controller	Y	blocked URL
O-RAN Central Unit		
O-RAN Distributed Unit High Layers	Y	blocked URL
O-RAN Distributed Unit Low Layers	Y	blocked URL
O-RAN Radio Unit		
Operations and Maintenance	Y	blocked URL
Simulations	Y	blocked URL
Infrastructure	Y	blocked URL
~~Integration and Testing~~	N/A	Cross-functional
~~Documentation~~	N/A	Cross-functional
Non-RealTime RIC (RAN Intelligent Controller)	Y	blocked URL
Service Management and Orchestration (SMO)		blocked URL

Projects that follow the best practices below can voluntarily self-certify and show that they've achieved a Core Infrastructure Initiative (CII) badge.

Instruction: Duplicate the list and complete the tables, as a child page for consideration. All contributors will have access to the CII listing (linked above) once created, and can update the criteria in the event a project team wishes to purse a higher grade (silver or gold). (Example project)

Criteria:

Basics (12 Points)

(Result/Proof point (column A: enter Met/Unmet; Column B: enter relevant URLs/comments)

	Project A	
Criteria	**Result / Proof point**	
Identification		
What is the human-readable name of the project?		
What is a brief description of the project?		
What is the URL for the project (as a whole)?		
What is the URL for the version control repository (it may be the same as the project URL)?		
What programming language(s) are used to implement the project?		
What is the Common Platform Enumeration (CPE) name for the project (if it has one)?		
Basic project website content		
The project website MUST succinctly describe what the software does (what problem does it solve?		
The project website MUST provide information on how to: obtain, provide feedback (as bug reports or enhancements), and contribute to the software.		
The information on how to contribute MUST explain the contribution process (e.g., are pull requests used?) (URL required)		
The information on how to contribute SHOULD include the requirements for acceptable contributions (e.g., a reference to any required coding standard). (URL required)		
FLOSS license		
What license(s) is the project released under?		

The software produced by the project MUST be released as FLOSS.		
It is SUGGESTED that any required license(s) for the software produced by the project be approved by the Open Source Initiative (OSI).		
The project MUST post the license(s) of its results in a standard location in their source repository.		
Documentation		
The project MUST provide basic documentation for the software produced by the project.		
The project MUST provide reference documentation that describes the external interface (both input and output) of the software produced by the project.		
Other		
The project sites (website, repository, and download URLs) MUST support HTTPS using TLS.		
The project MUST have one or more mechanisms for discussion (including proposed changes and issues) that are searchable, allow messages and topics to be addressed by URL, enable new people to participate in some of the discussions, and do not require client-side installation of proprietary software.		
The project SHOULD provide documentation in English and be able to accept bug reports and comments about code in English.		

Change Control (9 Points)

(Result/Proof point (column A: enter Met/Unmet; Column B: enter relevant URLs/comments)

	Project A	
Criteria	**Result / Proof point**	
Public version-controlled source repository		
The project MUST have a version-controlled source repository that is publicly readable and has a URL.		
The project's source repository MUST track what changes were made, who made the changes, and when the changes were made.		
To enable collaborative review, the project's source repository MUST include interim versions for review between releases; it MUST NOT include only final releases.		
It is SUGGESTED that common distributed version control software be used (e.g., git) for the project's source repository.		
Unique version numbering		
The project results MUST have a unique version identifier for each release intended to be used by users		
It is SUGGESTED that the Semantic Versioning (SemVer) format be used for releases.		
It is SUGGESTED that projects identify each release within their version control system. For example, it is SUGGESTED that those using git identify each release using git tags.		
Release notes		
The project MUST provide, in each release, release notes that are a human-readable summary of major changes in that release to help users determine if they should upgrade and what the upgrade impact will be. The release notes MUST NOT be the raw output of a version control log (e.g., the "git log" command results are not release notes). Projects whose results are not intended for reuse in multiple locations (such as the software for a single website or service) AND employ continuous delivery MAY select "N/A". (URL required)		
The release notes MUST identify every publicly known vulnerability with a CVE assignment or similar that is fixed in each new release, unless users typically cannot practically update the software themselves. If there are no release notes or there have been no publicly known vulnerabilities, choose "not applicable" (N/A).		

Reporting (8 Points)

(Result/Proof point (column A: enter Met/Unmet; Column B: enter relevant URLs/comments)

	Project A	
Criteria	**Result / Proof point**	
Bug-reporting process		
The project MUST provide a process for users to submit bug reports (e.g., using an issue tracker or a mailing list). (URL required)		
The project SHOULD use an issue tracker for tracking individual issues.		
The project MUST acknowledge a majority of bug reports submitted in the last 2-12 months (inclusive); the response need not include a fix.		
The project SHOULD respond to a majority (>50%) of enhancement requests in the last 2-12 months (inclusive).		
The project MUST have a publicly available archive for reports and responses for later searching. (URL required)		
Vulnerability report process		
The project MUST publish the process for reporting vulnerabilities on the project site. (URL required)		

If private vulnerability reports are supported, the project MUST include how to send the information in a way that is kept private. (URL required) Examples include a private defect report submitted on the web using HTTPS (TLS) or an email encrypted using OpenPGP. If vulnerability reports are always public (so there are never private vulnerability reports), choose "not applicable" (N/A).		
The project's initial response time for any vulnerability report received in the last 6 months MUST be less than or equal to 14 days. If there have been no vulnerabilities reported in the last 6 months, choose "not applicable" (N/A).		

Quality (13 Points)

(Result/Proof point (column A: enter Met/Unmet; Column B: enter relevant URLs/comments)

	Project A	
Criteria	**Result / Proof point**	
Working build system		
If the software produced by the project requires building for use, the project MUST provide a working build system that can automatically rebuild the software from source code.		
It is SUGGESTED that common tools be used for building the software.		
The project SHOULD be buildable using only FLOSS tools.		
Automated test suite		
The project MUST use at least one automated test suite that is publicly released as FLOSS (this test suite may be maintained as a separate FLOSS project).		
A test suite SHOULD be invocable in a standard way for that language. For example, "make check", "mvn test", or "rake test" (Ruby).		
It is SUGGESTED that the test suite cover most (or ideally all) the code branches, input fields, and functionality.		
It is SUGGESTED that the project implement continuous integration (where new or changed code is frequently integrated into a central code repository and automated tests are run on the result).		
New functionality testing		
The project MUST have a general policy (formal or not) that as major new functionality is added to the software produced by the project, tests of that functionality should be added to an automated test suite. As long as a policy is in place, even by word of mouth, that says developers should add tests to the automated test suite for major new functionality, select "Met.		
The project MUST have evidence that the test_policy for adding tests has been adhered to in the most recent major changes to the software produced by the project. Major functionality would typically be mentioned in the release notes. Perfection is not required, merely evidence that tests are typically being added in practice to the automated test suite when new major functionality is added to the software produced by the project.		
It is SUGGESTED that this policy on adding tests (see test_policy) be *documented* in the instructions for change proposals. However, even an informal rule is acceptable as long as the tests are being added in practice.		
Warning flags		
The project MUST enable one or more compiler warning flags, a "safe" language mode, or use a separate "linter" tool to look for code quality errors or common simple mistakes, if there is at least one FLOSS tool that can implement this criterion in the selected language.		
The project MUST address warnings.		
It is SUGGESTED that projects be maximally strict with warnings in the software produced by the project, where practical. Some warnings cannot be effectively enabled on some projects. What is needed is evidence that the project is striving to enable warning flags where it can, so that errors are detected early.		

Security (16 Points)

(Result/Proof point (column A: enter Met/Unmet; Column B: enter relevant URLs/comments)

	Project A	
Criteria	**Result / Proof point**	
Secure development knowledge		
The project MUST have at least one primary developer who knows how to design secure software. (See 'details' for the exact requirements.)		
At least one of the project's primary developers MUST know of common kinds of errors that lead to vulnerabilities in this kind of software, as well as at least one method to counter or mitigate each of them.		
Use basic good cryptographic practices		

The software produced by the project MUST use, by default, only cryptographic protocols and algorithms that are publicly published and reviewed by experts (if cryptographic protocols and algorithms are used).These cryptographic criteria do not always apply because some software has no need to directly use cryptographic capabilities.		
If the software produced by the project is an application or library, and its primary purpose is not to implement cryptography, then it SHOULD only call on software specifically designed to implement cryptographic functions; it SHOULD NOT re-implement its own.		
All functionality in the software produced by the project that depends on cryptography MUST be implementable using FLOSS. See the Open Standards Requirement for Software by the Open Source Initiative.		
The security mechanisms within the software produced by the project MUST use default keylengths that at least meet the NIST minimum requirements through the year 2030 (as stated in 2012). It MUST be possible to configure the software so that smaller keylengths are completely disabled.These minimum bitlengths are: symmetric key 112, factoring modulus 2048, discrete logarithm key 224, discrete logarithmic group 2048, elliptic curve 224, and hash 224 (password hashing is not covered by this bitlength, more information on password hashing can be found in the crypto_password_storage criterion). See https://www.keylength.com for a comparison of keylength recommendations from various organizations. The software MAY allow smaller keylengths in some configurations (ideally it would not, since this allows downgrade attacks, but shorter keylengths are sometimes necessary for interoperability).		
The default security mechanisms within the software produced by the project MUST NOT depend on broken cryptographic algorithms (e.g., MD4, MD5, single DES, RC4, Dual_EC_DRBG), or use cipher modes that are inappropriate to the context, unless they are necessary to implement an interoperable protocol (where the protocol implemented is the most recent version of that standard broadly supported by the network ecosystem, that ecosystem requires the use of such an algorithm or mode, and that ecosystem does not offer any more secure alternative). The documentation MUST describe any relevant security risks and any known mitigations if these broken algorithms or modes are necessary for an interoperable protocol.		
The default security mechanisms within the software produced by the project SHOULD NOT depend on cryptographic algorithms or modes with known serious weaknesses (e.g., the SHA-1 cryptographic hash algorithm or the CBC mode in SSH).		
The security mechanisms within the software produced by the project SHOULD implement perfect forward secrecy for key agreement protocols so a session key derived from a set of long-term keys cannot be compromised if one of the long-term keys is compromised in the future.		
If the software produced by the project causes the storing of passwords for authentication of external users, the passwords MUST be stored as iterated hashes with a per-user salt by using a key stretching (iterated) algorithm (e.g., Argon2id, Bcrypt, Scrypt, or PBKDF2). See also OWASP Password Storage Cheat Sheet).		
The security mechanisms within the software produced by the project MUST generate all cryptographic keys and nonces using a cryptographically secure random number generator, and MUST NOT do so using generators that are cryptographically insecure.		
Secured delivery against man-in-the-middle (MITM) attacks		
The project MUST use a delivery mechanism that counters MITM attacks. Using https or ssh+scp is acceptable.		
A cryptographic hash (e.g., a sha1sum) MUST NOT be retrieved over http and used without checking for a cryptographic signature.		
Publicly known vulnerabilities fixed		
There MUST be no unpatched vulnerabilities of medium or higher severity that have been publicly known for more than 60 days.		
Projects SHOULD fix all critical vulnerabilities rapidly after they are reported.		
Other security issues		
The public repositories MUST NOT leak a valid private credential (e.g., a working password or private key) that is intended to limit public access. A project MAY leak "sample" credentials for testing and unimportant databases, as long as they are not intended to limit public access.		

Analysis (8 Points)

(Result/Proof point (column A: enter Met/Unmet; Column B: enter relevant URLs/comments)

	Project A	
Criteria	**Result / Proof point**	
Static code analysis		
At least one static code analysis tool (beyond compiler warnings and "safe" language modes) MUST be applied to any proposed major production release of the software before its release, if there is at least one FLOSS tool that implements this criterion in the selected language.		
It is SUGGESTED that at least one of the static analysis tools used for the static_analysis criterion include rules or approaches to look for common vulnerabilities in the analyzed language or environment.		
All medium and higher severity exploitable vulnerabilities discovered with static code analysis MUST be fixed in a timely way after they are confirmed.		
It is SUGGESTED that static source code analysis occur on every commit or at least daily.		
Dynamic code analysis		
It is SUGGESTED that at least one dynamic analysis tool be applied to any proposed major production release of the software before its release.		
It is SUGGESTED that if the software produced by the project includes software written using a memory-unsafe language (e.g., C or C++), then at least one dynamic tool (e.g., a fuzzer or web application scanner) be routinely used in combination with a mechanism to detect memory safety problems such as buffer overwrites. If the project does not produce software written in a memory-unsafe language, choose "not applicable" (N/A).		
It is SUGGESTED that the software produced by the project include many run-time assertions that are checked during dynamic analysis.		
All medium and higher severity exploitable vulnerabilities discovered with dynamic code analysis MUST be fixed in a timely way after they are confirmed.		