CII Best Practices

Best Practices Criteria for Free/Libre and Open Source Software (FLOSS)

Introduction

This is a set of best practices for Free/Libre and Open Source Software (FLOSS) projects. Projects that follow these best practices will be able to voluntarily self-certify and show that they've achieved a Core Infrastructure Initiative (CII) badge. Projects can do this, at no cost, by using a web application (BadgeApp) to explain how they meet these practices and their detailed criteria.

There is no set of practices that can guarantee that software will never have defects or vulnerabilities; even formal methods can fail if the specifications or assumptions are wrong. Nor is there any set of practices that can guarantee that a project will sustain a healthy and well-functioning development community. However, following best practices can help improve the results of projects. For example, some practices enable multi-person review before release, which can both help find otherwise hard-to-find technical vulnerabilities and help build trust and a desire for repeated interaction among developers from different organizations.

These best practices have been created to:

1. encourage projects to follow best practices,
2. help new projects discover what those practices are, and
3. help users know which projects are following best practices (so users can prefer such projects).

The idiom "best practices" means "a procedure or set of procedures that is preferred or considered standard within an organization, industry, etc." (source: Dictionary.com). These criteria are what we believe are widely "preferred or considered standard" in the wider FLOSS community.

The "passing" criteria listed here focus on identifying best practices that well-run FLOSS projects typically already follow. The criteria are inspired by a variety of sources; see the separate "background" page for more information.

The criteria for higher/more advanced badges describe the criteria for the higher-level badges. These are known as the "silver" and "gold" levels, and sometimes also described as "other" criteria. You must achieve the "passing" criteria before you can achieve silver or gold.

The Linux Foundation also sponsors the OpenChain Project, which identifies criteria for a "high quality Free and Open Source Software (FOSS) compliance program." OpenChain focuses on how organizations can best use FLOSS and contribute back to FLOSS projects, while the CII Best Practices badge focuses on the FLOSS projects themselves. The CII Best Practices badge and OpenChain work together to help improve FLOSS and how FLOSS is used.

We expect that these practices and their detailed criteria will be updated, even after badges are released. Thus, criteria (and badges) probably will have a year identifier and will phase out after a year or two. We expect it will be easy to update the information, so this relatively short badge life should not be a barrier. We plan to add new criteria but mark them as "future" criteria, so that projects can add that information and maintain their badge.

Feedback is *very* welcome via the GitHub site as issues or pull requests. There is also a mailing list for general discussion.

Below are the current criteria, along with and where to get more information. The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as described in RFC 2119. The additional term SUGGESTED is added, as follows:

- The term MUST is an absolute requirement, and MUST NOT is an absolute prohibition.
- The term SHOULD indicates a criterion that is normally required, but there may exist valid reasons in particular circumstances to ignore it. However, the full implications must be understood and carefully weighed before choosing a different course.
- The term SUGGESTED is used instead of SHOULD when the criterion must be considered, but valid reasons to not do so are even more common than for SHOULD.
- Often a criterion is stated as something that SHOULD be done, or is SUGGESTED, because it may be difficult to implement or the costs to do so may be high.
- The term MAY provides one way something can be done, e.g., to make it clear that the described implementation is acceptable.
- To obtain a badge, all MUST and MUST NOT criteria must be met, all SHOULD criteria must be met OR the rationale for not implementing the criterion must be documented, and all SUGGESTED criteria have to be considered (rated as met or unmet). In some cases a URL may be required as part of the criterion's justification.
- The text "(Future criterion)" marks criteria that are not currently required, but may be required in the future.

We assume that you are already familiar with software development and running a FLOSS project; if not, see introductory materials such as *Producing Open Source Software* by Karl Fogel.

Terminology

A *project* is an active entity that has project member(s) and produces project result(s). Its member(s) use project sites to coordinate and disseminate result (s). A project does not need to be a formal legal entity. Key terms relating to project are:

- Project *members* are the group of one or more people or companies who work together to attempt to produce project results. Some FLOSS projects may have different kinds of members, with different roles, but that's outside our scope.
- Project *results* are what the project members work together to produce as their end goal. Normally this is software, but project results may include other things as well. Criteria that refer to "software produced by the project" are referring to project results.
- Project *sites* are the sites dedicated to supporting the development and dissemination of project results, and include the project website, repository, and download sites where applicable (see sites_https).

- The project *website*, aka project homepage, is the main page on the world wide web (WWW) that a new user would typically visit to see information about the project; it may be the same as the project's repository site (this is often true on GitHub).
- The project *repository* manages and stores the project results and revision history of the project results. This is also referred to as the project *source repository*, because we only require managing and storing of the editable versions, not of automatically generated results (in many cases generated results are not stored in a repository).

Current criteria: Best Practices for FLOSS

Here are the current criteria. Note that:

- Text inside square brackets is the short name of the criterion.
- In a few cases rationale is also included.
- We expect that there will be a few other fields for the project name, description, project URL, repository URL (which may be the same as the project URL), and license(s).
- In some cases N/A ("not applicable") may be an appropriate and permitted response.

In some cases we automatically test and fill in information if the project follows standard conventions and is hosted on a site (e.g., GitHub) with decent API support. We intend to improve this automation in the future (improvements welcome!).

The actual criteria are stored in the file "criteria/criteria.yml", including details, rationale, and how it could be automated.

There is an implied criterion that we should mention here:

- The project MUST have a public website with a stable URL. (The badging application enforces this by requiring a URL to create a badge entry.) [homepage_url]

Basics

Basic project website content

- The project website MUST succinctly describe what the software does (what problem does it solve?). [description_good]

  Details:This MUST be in language that potential users can understand (e.g., it uses minimal jargon).
- The project website MUST provide information on how to: obtain, provide feedback (as bug reports or enhancements), and contribute to the software. [interact]
- The information on how to contribute MUST explain the contribution process (e.g., are pull requests used?) (URL required for "met".) [contribution]

  Details:We presume that projects on GitHub use issues and pull requests unless otherwise noted. This information can be short, e.g., stating that the project uses pull requests, an issue tracker, or posts to a mailing list (which one?)Rationale:Contributors need to understand not only how to contribute, but also the overall contribution process, so that they'll understand how their work could be incorporated and what the expectations are after the initial submission. This means that wherever the project describes how to contribute, the project must include (directly or by reference) information on the contribution process. Note that criterion "interact" (listed earlier) requires that the contribution information be on the project website.
- The information on how to contribute SHOULD include the requirements for acceptable contributions (e.g., a reference to any required coding standard). (URL required for "met".) [contribution_requirements]

OSS License

- The software produced by the project MUST be released as FLOSS. [floss_license]

  Details:FLOSS is software released in a way that meets the Open Source Definition or Free Software Definition. Examples of such licenses include the CC0, MIT, BSD 2-clause, BSD 3-clause revised, Apache 2.0, Lesser GNU General Public License (LGPL), and the GNU General Public License (GPL). For our purposes, this means that the license MUST be:an approved license by the Open Source Initiative (OSI), ora free license as approved by the Free Software Foundation (FSF), ora free license acceptable to Debian main, ora "good" license according to Fedora. The software MAY also be licensed other ways (e.g., "GPLv2 or proprietary" is acceptable).Rationale:These criteria are designed for FLOSS projects, so we need to ensure that they're only used where they apply. Some projects may be mistakenly considered FLOSS even though they are not (e.g., they might not have any license, in which case the defaults of the country's legal system apply, or they might use a non-FLOSS license). We've added "produced by the project" as a clarification - many projects use non-FLOSS software/services in the process of creating software, or depend on them to run, and that is allowed.
- It is SUGGESTED that any required license(s) for the software produced by the project be approved by the Open Source Initiative (OSI). [floss_license_osi]

  Details:The OSI uses a rigorous approval process to determine which licenses are OSS.Rationale:Unusual licenses can cause long-term problems for FLOSS projects and are more difficult for tools to handle. That said, there are FLOSS licenses that are not OSI-approved, e.g., the CC0 license is used by many projects but is not OSI-approved at the time of this writing. We expect that more advanced badges would set a higher bar (e.g., that it must be released under an OSI-approved license).
- The project MUST post the license(s) of its results in a standard location in their source repository. (URL required for "met".) [license_location]

Details:E.g., as a top-level file named LICENSE or COPYING. License filenames MAY be followed by an extension such as ".txt" or ".md". Note that this criterion is only a requirement on the source repository. You do NOT need to include the license file when generating something from the source code (such as an executable, package, or container). For example, when generating an R package for the Comprehensive R Archive Network (CRAN), follow standard CRAN practice: if the license is a standard license, use the standard short license specification (to avoid installing yet another copy of the text) and list the LICENSE file in an exclusion file such as .Rbuildignore. Similarly, when creating a Debian package, you may put a link in the copyright file to the license text in /usr/share/common-licenses, and exclude the license file from the created package (e.g., by deleting the file after calling dh_auto_install). We do encourage including machine-readable license information in generated formats where practical.Rationale:The goal is to make the license very clear and connected with the project results it applies to. It is a good idea to also make the license clear on the project website, but there isn't a widely-accepted way to do that today.

Documentation

- The project MUST provide basic documentation for the software produced by the project. (N/A allowed.) (Justification required for "N/A".) [documentation_basics]

  Details:This documentation must be in some media (such as text or video) that includes: how to install it, how to start it, how to use it (possibly with a tutorial using examples), and how to use it securely (e.g., what to do and what not to do) if that is an appropriate topic for the software. The security documentation need not be long. The project MAY use hypertext links to non-project material as documentation. If the project does not produce software, choose "not applicable" (N/A).Rationale:Potential users need documentation so that they can learn how to use the software. This documentation could be provided via the project website or repository, or even via hyperlink to some external information, so we do not specify exactly where this information is.
- The project MUST provide reference documentation that describes the external interface (both input and output) of the software produced by the project. (N/A allowed.) (Justification required for "N/A".) [documentation_interface]

  Details:The documentation of an external interface explains to an end-user or developer how to use it. This would include its application program interface (API) if the software has one. If it is a library, document the major classes/types and methods/functions that can be called. If it is a web application, define its URL interface (often its REST interface). If it is a command-line interface, document the parameters and options it supports. In many cases it's best if most of this documentation is automatically generated, so that this documentation stays synchronized with the software as it changes, but this isn't required. The project MAY use hypertext links to non-project material as documentation. Documentation MAY be automatically generated (where practical this is often the best way to do so). Documentation of a REST interface may be generated using Swagger/OpenAPI. Code interface documentation MAY be generated using tools such as JSDoc (JavaScript), ESDoc (JavaScript), pydoc (Python), devtools (R), pkgdown (R), and Doxygen (many). Merely having comments in implementation code is not sufficient to satisfy this criterion; there needs to be an easy way to see the information without reading through all the source code. If the project does not produce software, choose "not applicable" (N/A).

Other

- The project sites (website, repository, and download URLs) MUST support HTTPS using TLS. [sites_https]

  Details:This requires that the project home page URL and the version control repository URL begin with "https:", not "http:". You can get free certificates from Let's Encrypt. Projects MAY implement this criterion using (for example) GitHub pages, GitLab pages, or SourceForge project pages. If you are using GitHub pages with custom domains, you MAY use a content delivery network (CDN) as a proxy to support HTTPS, such as described in the blog post "Secure and fast GitHub Pages with CloudFlare", to satisfy this criterion. If you support HTTP, we urge you to redirect the HTTP traffic to HTTPS.
- 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. [discussion]

  Details:Examples of acceptable mechanisms include archived mailing list(s), GitHub issue and pull request discussions, Bugzilla, Mantis, and Trac. Asynchronous discussion mechanisms (like IRC) are acceptable if they meet these criteria; make sure there is a URL-addressable archiving mechanism. Proprietary JavaScript, while discouraged, is permitted.
- The project SHOULD provide documentation in English and be able to accept bug reports and comments about code in English. [english]

  Details:English is currently the lingua franca of computer technology; supporting English increases the number of different potential developers and reviewers worldwide. A project can meet this criterion even if its core developers' primary language is not English.

Change Control

Public version-controlled source repository

- The project MUST have a version-controlled source repository that is publicly readable and has a URL. [repo_public]

  Details:The URL MAY be the same as the project URL. The project MAY use private (non-public) branches in specific cases while the change is not publicly released (e.g., for fixing a vulnerability before it is revealed to the public).
- The project's source repository MUST track what changes were made, who made the changes, and when the changes were made. [repo_track]
- To enable collaborative review, the project's source repository MUST include interim versions for review between releases; it MUST NOT include only final releases. [repo_interim]

  Details:Projects MAY choose to omit specific interim versions from their public source repositories (e.g., ones that fix specific non-public security vulnerabilities, may never be publicly released, or include material that cannot be legally posted and are not in the final release).
- It is SUGGESTED that common distributed version control software be used (e.g., git) for the project's source repository. [repo_distributed]

  Details:Git is not specifically required and projects can use centralized version control software (such as subversion) with justification.

Unique version numbering

- The project results MUST have a unique version identifier for each release intended to be used by users. [version_unique]

  Details:This MAY be met in a variety of ways including a commit IDs (such as git commit id or mercurial changeset id) or a version number (including version numbers that use semantic versioning or date-based schemes like YYYYMMDD).
- It is SUGGESTED that the Semantic Versioning (SemVer) format be used for releases. [version_semver]

  Details:Other version numbering schemes, such as commit IDs (such as git commit id or mercurial changeset id) or date-based schemes like YYYYMMDD, MAY be used as version numbers, since they are unique. Some alternatives can cause problems, because users may not be able to easily determine if they are up-to-date. SemVer may be less helpful for identifying software releases if all recipients only run the latest version (e.g., it is the code for a single website or internet service that is constantly updated via continuous delivery).Rationale:SemVer is widely used to communicate what an update is (e.g., if it involves incompatible API changes), whether something is newer or older. The scheme is simple, supports multiple simultaneous branches, and because it uses at least three numbers it can be distinguished from floating point. However, many find SemVer less useful for identifying software versions if only one version of the component is run (e.g., it is the code for a single website or internet service that is constantly updated via continuous delivery). For more discussion of the pros and cons of SemVer, see Hacker News' Is Semantic Versioning an Anti-Pattern? and The Semantic Versioning Anti-Pattern.
- 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. [version_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". (N/A allowed.) (Justification required for "N/A".) (URL required for "met".) [release_notes]

  Details:The release notes MAY be implemented in a variety of ways. Many projects provide them in a file named "NEWS", "CHANGELOG", or "ChangeLog", optionally with extensions such as ".txt", ".md", or ".html". Historically the term "change log" meant a log of every change, but to meet these criteria what is needed is a human-readable summary. The release notes MAY instead be provided by version control system mechanisms such as the GitHub Releases workflow.Rationale:Release notes are important because they help users decide whether or not they will want to update, and what the impact would be (e.g., if the new release fixes vulnerabilities). We realize this may not apply to projects whose main results are continuously updated and are deployed to primarily one place and so allow "N/A" from such projects.
- 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). (N/A allowed.) (Justification required for "N/A".) [release_notes_vulns]

  Details:If users typically cannot practically update the software themselves on their computers, but must instead depend on a middleman to perform the upgrade (as is often the case for a kernel and low-level software that is intertwined with a kernel), the project may choose "not applicable" (N/A).

Reporting

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 for "met".) [report_process]
- The project SHOULD use an issue tracker for tracking individual issues. [report_tracker]
- The project MUST acknowledge a majority of bug reports submitted in the last 2-12 months (inclusive); the response need not include a fix. [report_responses]
- The project SHOULD respond to a majority (>50%) of enhancement requests in the last 2-12 months (inclusive). [enhancement_responses]

  Details:The response MAY be 'no' or a discussion about its merits. The goal is simply that there be some response to some requests, which indicates that the project is still alive. For purposes of this criterion, projects need not count fake requests (e.g., from spammers or automated systems). If a project is no longer making enhancements, please select "unmet" and include the URL that makes this situation clear to users. If a project tends to be overwhelmed by the number of enhancement requests, please select "unmet" and explain.
- The project MUST have a publicly available archive for reports and responses for later searching. (URL required for "met".) [report_archive]

Vulnerability report process

- The project MUST publish the process for reporting vulnerabilities on the project site. (URL required for "met".) [vulnerability_report_process]

  Details:E.g., a clearly designated mailing address on https://PROJECTSITE/security, often in the form security@example.org. This MAY be the same as its bug reporting process. Vulnerability reports MAY always be public, but many projects have a private vulnerability reporting mechanism.
- If private vulnerability reports are supported, the project MUST include how to send the information in a way that is kept private. (N/A allowed.) (URL required for "met".) [vulnerability_report_private]

  Details: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. (N/A allowed.) [vulnerability_report_response]

  Details:If there have been no vulnerabilities reported in the last 6 months, choose "not applicable" (N/A).

Quality

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. (N/A allowed.) [build]

  Details:A build system determines what actions need to occur to rebuild the software (and in what order), and then performs those steps. For example, it can invoke a compiler to compile the source code. If an executable is created from source code, it must be possible to modify the project's source code and then generate an updated executable with those modifications. If the software produced by the project depends on external libraries, the build system does not need to build those external libraries. If there is no need to build anything to use the software after its source code is modified, select "not applicable" (N/A).Rationale:If a project needs to be built but there is no working build system, then potential co-developers will not be able to easily contribute and many security analysis tools will be ineffective. This is related to Joel Test point 2, "Can you make a build in one step?"
- It is SUGGESTED that common tools be used for building the software. (N/A allowed.) [build_common_tools]

  Details:For example, Maven, Ant, cmake, the autotools, make, rake (Ruby), or devtools (R).
- The project SHOULD be buildable using only FLOSS tools. (N/A allowed.) [build_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). [test]

  Details:The project MAY use multiple automated test suites (e.g., one that runs quickly, vs. another that is more thorough but requires special equipment). There are many test frameworks and test support systems available, including Selenium (web browser automation), Junit (JVM, Java), RUnit (R), testthat (R).Rationale:Automated test suites immediately help detect a variety of problems. A large test suite can find more problems, but even a small test suite can detect problems and provide a framework to build on. E.g., "Tip #62: Test Early, Test Often, Test Automatically" ("The Pragmatic Programmer" by Andrew Hunt and David Thomas, p. 237)
- A test suite SHOULD be invocable in a standard way for that language. [test_invocation]

  Details: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. [test_most]
- 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). [test_continuous_integration]

  Rationale:See Martin Fowler There has been some shift in the meaning of the term continuous integration. Historically the term continuous integration focused on the first part - the frequent integration - and not on its testing. However, over time the emphasis has shifted to include the notion of running automated tests as soon as the code is integrated. We realize that this can be difficult for some projects to apply, which is why it is only SUGGESTED at the passing level.

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. [test_policy]

  Details: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. [tests_are_added]

  Details: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. [tests_documented_added]

  Details: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. (N/A allowed.) [warnings]

  Details:Examples of compiler warning flags include gcc/clang "-Wall". Examples of a "safe" language mode include JavaScript "use strict" and perl5's "use warnings". A separate "linter" tool is simply a tool that examines the source code to look for code quality errors or common simple mistakes. These are typically enabled within the source code or build instructions.Rationale:"We routinely set compiler warning levels as high as possible. It doesn't make sense to waste time trying to find a problem that the compiler could find for you! We need to concentrate on the harder problems at hand." ("The Pragmatic Programmer" by Andrew Hunt and David Thomas, p. 91-92) "Tip #23: Always use Source Code Control. Always. Even if you are a single-person team on a one-week project." ("The Pragmatic Programmer" by Andrew Hunt and David Thomas, p. 88)
- The project MUST address warnings. (N/A allowed.) [warnings_fixed]

  Details:These are the warnings identified by the implementation of the warnings criterion. The project should fix warnings or mark them in the source code as false positives. Ideally there would be no warnings, but a project MAY accept some warnings (typically less than 1 warning per 100 lines or less than 10 warnings).
- It is SUGGESTED that projects be maximally strict with warnings in the software produced by the project, where practical. (N/A allowed.) [warnings_strict]

Details: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

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.) [know_secure_design]

  Details:This requires understanding the following design principles, including the 8 principles from Saltzer and Schroeder:economy of mechanism (keep the design as simple and small as practical, e.g., by adopting sweeping simplifications)fail-safe defaults (access decisions should deny by default, and projects' installation should be secure by default)complete mediation (every access that might be limited must be checked for authority and be non-bypassable)open design (security mechanisms should not depend on attacker ignorance of its design, but instead on more easily protected and changed information like keys and passwords)separation of privilege (ideally, access to important objects should depend on more than one condition, so that defeating one protection system won't enable complete access. E.G., multi-factor authentication, such as requiring both a password and a hardware token, is stronger than single-factor authentication)least privilege (processes should operate with the least privilege necessary)least common mechanism (the design should minimize the mechanisms common to more than one user and depended on by all users, e.g., directories for temporary files)psychological acceptability (the human interface must be designed for ease of use - designing for "least astonishment" can help)limited attack surface (the attack surface - the set of the different points where an attacker can try to enter or extract data - should be limited)input validation with whitelists (inputs should typically be checked to determine if they are valid before they are accepted; this validation should use whitelists (which only accept known-good values), not blacklists (which attempt to list known-bad values)).A "primary developer" in a project is anyone who is familiar with the project's code base, is comfortable making changes to it, and is acknowledged as such by most other participants in the project. A primary developer would typically make a number of contributions over the past year (via code, documentation, or answering questions). Developers would typically be considered primary developers if they initiated the project (and have not left the project more than three years ago), have the option of receiving information on a private vulnerability reporting channel (if there is one), can accept commits on behalf of the project, or perform final releases of the project software. If there is only one developer, that individual is the primary developer.
- 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. [know_common_errors]

  Details:Examples (depending on the type of software) include SQL injection, OS injection, classic buffer overflow, cross-site scripting, missing authentication, and missing authorization. See the CWE/SANS top 25 or OWASP Top 10 for commonly used lists.

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). (N/A allowed.) [crypto_published]

  Details: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. (N/A allowed.) [crypto_call]
- All functionality in the software produced by the project that depends on cryptography MUST be implementable using FLOSS. (N/A allowed.) [crypto_floss]

  Details:See the Open Standards Requirement for Software by the Open Source Initiative.Rationale:Software must interoperate with other software. If the functionality cannot be implemented with FLOSS, e.g., because of patents, then this can set a trap for others who depend on the software.
- 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. (N/A allowed.) [crypto_keylength]

  Details: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 http://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. (N/A allowed.) [crypto_working]

  Details:ECB mode is almost never appropriate because it reveals identical blocks within the ciphertext as demonstrated by the ECB penguin, and CTR mode is often inappropriate because it does not perform authentication and causes duplicates if the input state is repeated. In many cases it's best to choose a block cipher algorithm mode designed to combine secrecy and authentication, e.g., Galois/Counter Mode (GCM) and EAX. Projects MAY allow users to enable broken mechanisms (e.g., during configuration) where necessary for compatibility, but then users know they're doing it.Rationale:If a cryptographic algorithm or mode is completely broken, then it cannot provide a useful cryptographic service. This is different from having a weakness; many cryptographic algorithms have some weaknesses, yet for backwards-compatibility it may sometimes be appropriate to use the algorithm anyway. "EAX" appears to be a name, not an abbrevation. The paper describing EAX, "A Conventional Authenticated-Encryption Mode" by M. Bellare, P. Rogaway D. Wagner (April 13, 2003), does not give an expansion.
- 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). (N/A allowed.) [crypto_weaknesses]

Details:Concerns about CBC mode in SSH are discussed in CERT: SSH CBC vulnerability.Rationale:SHA-1 has been known to be weak for many years; In February 2017 Google demonstrated a SHA-1 collision. There are a number of alternatives to SHA-1 that are not patent-encumbered, such as the SHA-2 suite (including SHA-256 and SHA-512) and SHA-3. There is some disagreement on how important it is to avoid CBC mode in SSH. The OpenSSH cbc.adv page argues that the attack on SSH CBC is not a practical attack. However, others clearly think it's more important; CERT notes it, as does FAQ: Disable CBC in SSH. It is also easy to use a different mode than CBC; generally when there are safer widely-available options, you should use the safe ones instead. This is a SHOULD, not a MUST; sometimes these weaker mechanisms need to be used for backwards compatibility.

- 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. (N/A allowed.) [crypto_pfs]
- 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., PBKDF2, Bcrypt or Scrypt). (N/A allowed.) [crypto_password_storage]

  Details:This criterion applies only when the software is enforcing authentication of users using passwords, such as server-side web applications. It does not apply in cases where the software stores passwords for authenticating into other systems (e.g., the software implements a client for some other system), since at least parts of that software must have often access to the unhashed password.Rationale:This is a bare minimum today when storing passwords. Sometimes software needs to have a credential, such as a password, to authenticate it to other systems; those are intentionally out of scope for this criterion, because in many cases it's not possible to store them as iterated hashes using per-user salt.
- 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. (N/A allowed.) [crypto_random]

  Details:A cryptographically secure random number generator may be a hardware random number generator, or it may be a cryptographically secure pseudo-random number generator (CSPRNG) using an algorithm such as Hash_DRBG, HMAC_DRBG, CTR_DRBG, Yarrow, or Fortuna. Examples of calls to secure random number generators include Java's java.security.SecureRandom and JavaScript's window.crypto.getRandomValues. Examples of calls to insecure random number generators include Java's java.util.Random and JavaScript's Math.random.

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. [delivery_mitm]

  Details:An even stronger mechanism is releasing the software with digitally signed packages, since that mitigates attacks on the distribution system, but this only works if the users can be confident that the public keys for signatures are correct and if the users will actually check the signature.
- A cryptographic hash (e.g., a sha1sum) MUST NOT be retrieved over http and used without checking for a cryptographic signature. [delivery_unsigned]

  Details:These hashes can be modified in transit.

Publicly known vulnerabilities fixed

- There MUST be no unpatched vulnerabilities of medium or high severity that have been publicly known for more than 60 days. [vulnerabilities_fixed_60_days]

  Details:The vulnerability must be patched and released by the project itself (patches may be developed elsewhere). A vulnerability becomes publicly known (for this purpose) once it has a CVE with publicly released non-paywalled information (reported, for example, in the National Vulnerability Database) or when the project has been informed and the information has been released to the public (possibly by the project). A vulnerability is medium to high severity if its CVSS 2.0 base score is 4 or higher. Note: this means that users might be left vulnerable to all attackers worldwide for up to 60 days. This criterion is often much easier to meet than what Google recommends in Rebooting responsible disclosure, because Google recommends that the 60-day period start when the project is notified even if the report is not public. Also note that this badge criterion, like other criteria, applies to the individual project. Some projects are part of larger umbrella organizations or larger projects, possibly in multiple layers, and many projects feed their results to other organizations and projects as part of a potentially-complex supply chain. An individual project often cannot control the rest, but an individual project can work to release a vulnerability patch in a timely way. Therefore, we focus solely on the individual project's response time. Once a patch is available from the individual project, others can determine how to deal with the patch (e.g., they can update to the newer version or they can apply just the patch as a cherry-picked solution).Rationale:We intentionally chose to start measurement from the time of public knowledge, and not from the time reported to the project, because this is much easier to measure and verify by those outside the project.
- Projects SHOULD fix all critical vulnerabilities rapidly after they are reported. [vulnerabilities_critical_fixed]

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. [no_leaked_credentials]

  Details:A project MAY leak "sample" credentials for testing and unimportant databases, as long as they are not intended to limit public access.

Analysis

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. (N/A allowed.) (Justification required for "N/A".) [static_analysis]

Details:A static code analysis tool examines the software code (as source code, intermediate code, or executable) without executing it with specific inputs. For purposes of this criterion, compiler warnings and "safe" language modes do not count as static code analysis tools (these typically avoid deep analysis because speed is vital). Some static analysis tools focus on detecting generic defects, others focus on finding specific kinds of defects (such as vulnerabilities), and some do a combination. Examples of such static code analysis tools include cppcheck (C, C++), clang static analyzer (C, C++), SpotBugs (Java), FindBugs (Java) (including FindSecurityBugs), PMD (Java), Brakeman (Ruby on Rails), lintr (R), goodpractice (R), Coverity Quality Analyzer, SonarQube, Codacy, and HP Enterprise Fortify Static Code Analyzer. Larger lists of tools can be found in places such as the Wikipedia list of tools for static code analysis, OWASP information on static code analysis, NIST list of source code security analyzers, and Wheeler's list of static analysis tools. The SWAMP is a no-cost platform for assessing vulnerabilities in software using a variety of tools. If there are no FLOSS static analysis tools available for the implementation language(s) used, select 'N/A'.

- 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. (N/A allowed.) [static_analysis_common_vulnerabilities]

  Details:Static analysis tools that are specifically designed to look for common vulnerabilities are more likely to find them. That said, using any static tools will typically help find some problems, so we are suggesting but not requiring this for the 'passing' level badge.Rationale:We'd like all projects to use this kind of static analysis tool, but there may not be one in the chosen language, or it may only be proprietary (and some developers will therefore not use it).
- All medium and high severity exploitable vulnerabilities discovered with static code analysis MUST be fixed in a timely way after they are confirmed. (N/A allowed.) [static_analysis_fixed]

  Details:A vulnerability is medium to high severity if its CVSS 2.0 is 4 or higher.
- It is SUGGESTED that static source code analysis occur on every commit or at least daily. (N/A allowed.) [static_analysis_often]

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. [dynamic_analysis]

  Details:A dynamic analysis tool examines the software by executing it with specific inputs. For example, the project MAY use a fuzzing tool (e.g., American Fuzzy Lop) or a web application scanner (e.g., OWASP ZAP or w3af). In some cases the OSS-Fuzz project may be willing to apply fuzz testing to your project. For purposes of this criterion the dynamic analysis tool needs to vary the inputs in some way to look for various kinds of problems or be an automated test suite with at least 80% branch coverage. The Wikipedia page on dynamic analysis and the OWASP page on fuzzing identify some dynamic analysis tools. The analysis tool(s) MAY be focused on looking for security vulnerabilities, but this is not required. Rationale:Static source code analysis and dynamic analysis tend to find different kinds of defects (including defects that lead to vulnerabilities), so combining them is more likely to be effective. For example, Linus Torvalds' "Linux 4.14-rc5" announcement (October 15, 2017) notes that "(people are doing) random fuzzing... and it's finding things... Very nice to see."
- 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). (N/A allowed.) [dynamic_analysis_unsafe]

  Details:Examples of mechanisms to detect memory safety problems include Address Sanitizer (ASAN) (available in GCC and LLVM), Memory Sanitizer, and valgrind. Other potentially-used tools include thread sanitizer and undefined behavior sanitizer. Widespread assertions would also work.
- It is SUGGESTED that the software produced by the project include many run-time assertions that are checked during dynamic analysis. [dynamic_analysis_enable_assertions]

  Rationale:Assertions make dynamic analysis more effective, because they increase the number of problems (including vulnerabilities) that dynamic analysis can detect. Other sources also recommend the use of assertions. "Tip #33: If it Can't happen, use assertions to ensure that it won't." ("The Pragmatic Programmer" by Andrew Hunt and David Thomas, p. 122) The paper "Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation" by Gunnar Kudrjavets, Nachi Nagappan, and Tom Ball, May 1, 2006, Technical report MSR-TR-2006-54, presented "... an empirical case study of two commercial software components at Microsoft Corporation. The developers of these components systematically employed assertions, which allowed us to investigate the relationship between software assertions and code quality... with an increase in the assertion density in a file there is a statistically significant decrease in fault density. Further, the usage of software assertions in these components found a large percentage of the faults in the bug database."
- All medium and high severity exploitable vulnerabilities discovered with dynamic code analysis MUST be fixed in a timely way after they are confirmed. (N/A allowed.) [dynamic_analysis_fixed]

  Details:A vulnerability is medium to high severity if its CVSS 2.0 base score is 4. If you are not running dynamic code analysis and thus have not found any vulnerabilities in this way, choose "not applicable" (N/A).

A note on good cryptographic practices

Note: These criteria do not always apply because some software has no need to directly use cryptographic capabilities. A "project security mechanism" is a security mechanism provided by the delivered project's software.

Non-criteria

We do *not* require any specific products or services, and in general do not require any particular technology. In particular, we do *not* require proprietary tools, services, or technology, since many free software developers would reject such criteria. For example, we intentionally do *not* require git or GitHub. We also do not require or forbid any particular programming language. We do require that additional measures be taken for certain *kinds* of programming languages, but that is different. This means that as new tools and capabilities become available, projects can quickly switch to them without failing to meet any criteria.

We *do* provide guidance and help for common cases. The criteria *do* sometimes identify common methods or ways of doing something (especially if they are FLOSS), since that information can help people understand and meet the criteria. We also created an "easy on-ramp" for projects using git on GitHub, since that is a common case. But note that nothing *requires* git or GitHub. We would welcome good patches that help provide an "easy on-ramp" for projects on other repository platforms; GitLab was one of the first projects with a badge.

We avoid requiring, at the passing level, criteria that would be impractical for a single-person project, e.g., something that requires a significant amount of money. Many FLOSS projects are small, and we do not want to disenfranchise them.

We do not plan to require active user discussion within a project. Some highly mature projects rarely change and thus may have little activity. We *do*, however, require that the project be responsive if vulnerabilities are reported to the project (see above).

Uniquely identifying a project

One challenge is uniquely identifying a project. Our Rails application gives a unique id to each new project, so we can use that id to uniquely identify project entries. However, that doesn't help people who searching for the project and do not already know that id.

The *real* name of a project, for our purposes, is the URL for its repository, and where that is not available, the project "front page" URL can help find it. Most projects have a human-readable name, and we provide a search mechanims, but these names are not enough to uniquely identify a project. The same human-readable name can be used for many different projects (including project forks), and the same project may go by many different names. In many cases it will be useful to point to other names for the project (e.g., the source package name in Debian, the package name in some language-specific repository, or its name in OpenHub).

In the future we may try to check more carefully that a user can legitimately represent a project. For the moment, we primarily focus on checking if GitHub repositories are involved; there are ways to do this for other situations if that becomes important.

Non-admin users cannot edit the repo URL once one is entered. (Exception: they can upgrade http to https.) If they could change the repo URL, they might fool people into thinking they controlled a project that they did not. That said, creating a bogus row entry does not really help someone very much; what matters to the software is the id used by the project when it refers to its badge, and the project determines that.

Why have criteria?

The paper Open badges for education: what are the implications at the intersection of open systems and badging?identifies three general reasons for badging systems (all are valid for this):

1. Badges as a motivator of behavior. We hope that by identifying best practices, we'll encourage projects to implement those best practices if they do not do them already.
2. Badges as a pedagogical tool. Some projects may not be aware of some of the best practices applied by others, or how they can be practically applied. The badge will help them become aware of them and ways to implement them.
3. Badges as a signifier or credential. Potential users want to use projects that are applying best practices to consistently produce good results; badges make it easy for projects to signify that they are following best practices, and make it easy for users to see which projects are doing so.

We have chosen to use self-certification, because this makes it possible for a large number of projects (even small ones) to participate. There's a risk that projects may make false claims, but we think the risk is small, and users can check the claims for themselves.

Improving the criteria

We are hoping to get good suggestions and feedback from the public; please contribute!

We launched with a single badge level called *passing*. For higher level badges, see other.

You may also want to see the "background" file for more information about these criteria, and the "implementation" notes about the BadgeApp application.

See also

Project participation and interface:

- CONTRIBUTING.md - How to contribute to this project
- INSTALL.md - How to install/quick start
- governance.md - How the project is governed
- roadmap.md - Overall direction of the project
- background.md - Background research
- api - Application Programming Interface (API), inc. data downloads

Criteria:

- criteria.md - Criteria for "passing" badge
- other.md - Criteria for other badges (silver and gold)

Development processes and security:

- requirements.md - Requirements (what's it supposed to do?)
- design.md - Architectural design information
- implementation.md - Implementation notes
- testing.md - Information on testing

- security.md - Why it's adequately secure (assurance case)

- security.md - Why it's adequately secure (assurance case)