PREMIÈRE MINISTRE

Liberté
Égalité
Fraternité

Secrétariat général de la défense et de la sécurité nationale

Agence nationale de la sécurité des systèmes d'information

Paris, le 27 septembre 2022

N° 2089

Référence : ANSSI-CC-NOTE-26_v1.0

NOTE D'APPLICATION

Exigences pour l'analyse statique outillée de code lors des évaluations de sécurité

Application : Dès son approbation.

Diffusion : Publique.

Le Sous-directeur « Expertise »
de l'agence nationale de la sécurité
des systèmes d'information

Renaud LABELLE
[ORIGNAL SIGNE]

51 boulevard de La Tour-Maubourg - 75700 PARIS 07 SP - Tél 01.71.75.82.82

SUIVI DES MODIFICATIONS

Version	Date	Modifications
1.0		Première version

En application du décret n°2002-535 du 18 avril 2002 modifié, la présente note a été soumise, lors de sa création, au comité directeur de la certification, qui a donné un avis favorable.

Cette note est également soumise pour avis lors de chaque modification majeure conformément au manuel qualité du centre de certification. Les évolutions mineures ne sont pas soumises au comité directeur de la certification.

La présente note est disponible en ligne sur le site institutionnel de l'ANSSI (www.ssi.gouv.fr).

TABLE DES MATIERES

1 Objet de la note et applicabilité 4
1.1 Rôle du code source dans l'analyse de vulnérabilité 4
2 Exigences pour le commanditaire 5
3 Exigences pour l'évaluateur 7
ANNEXE A. Table of contents for code analysis methods 9
1. Introduction 9
2. Preliminary steps 9
2.1. Inputs from the developer 9
2.2. Analysis of the compilation 10
3. Analysis tool selection 10
3.1. List of static analysis tools used by the evaluator 10
3.2. Identification of constraints 11
3.3. Recommendations 11
4. Analysis tool settings 11
5. Code adaptation for analysis 11
5.1. General principles 11
5.2. Guidance for code adaptation 12
6. First run/test step 12
7. Real code analyses 12
8. Review of tool results 13
8.1. General method and interpretation guidelines 13
8.2. About the possible fixes to improve analysis 14
ANNEXE B. Références 15

1 Objet de la note et applicabilité

Cette note est applicable aux évaluations selon les Critères Communs demandant la conformité aux exigences AVA_VAN.3 ou supérieures.

Dans cette note, le terme de « code source » désigne uniquement les sources écrites dans des langages de programmation compilés ou interprétés (par exemple C, Java, Python ou PHP), par opposition aux langages de description de matériel (par exemple VHDL ou Verilog) qui sont hors périmètre de cette note.

1.1 Rôle du code source dans l'analyse de vulnérabilité

La raison d'être de cette note provient des exigences AVA_VAN.3.3E/AVA_VAN.4.3E/AVA_VAN.5.3E. Ces exigences demandent à l'évaluateur d'effectuer une analyse de vulnérabilités de la TOE en prenant en compte les guides du produit, la spécification fonctionnelle, la documentation de conception, l'architecture de sécurité et l'implémentation, afin d'identifier des vulnérabilités potentielles dans la TOE (« *a vulnerability analysis of the TOE using the guidance documentation, functional specification, TOE design, security architecture description and implementation representation to identify potential vulnerabilities in the TOE »).*

L'implémentation ou code source est nécessaire à ce stade pour identifier deux types différents de vulnérabilités :

- les vulnérabilités logiques, propres au type de produit évalué, qui peuvent seulement être vérifiées au niveau du code source ;

  EXEMPLE :

  - sur une carte à puce, le manque de contremesure protégeant la TOE contre les injections de fautes,

  - dans une application, la présence d'information résiduelle, par exemple un mot de passe en clair dans le code.

- les vulnérabilités introduites par un mauvais usage, ou une limitation intrinsèque, de la technologie d'implémentation (langage de programmation, compilateur…) ;

  EXEMPLE : dans un programme écrit en C,

  - un dépassement de tampon dû à l'utilisation de strcpy,

  - une clé secrète laissée en mémoire car sa mise à zéro a été supprimée par le compilateur (en raison d'options d'optimisation à la compilation).

Cette note concerne uniquement la recherche des dernières vulnérabilités, c'est-à-dire celles introduites par un mauvais usage ou une limitation intrinsèque de la technologie d'implémentation.

L'exigence principale de cette note vient du retour d'expérience du centre de certification au sujet de l'analyse statique manuelle[1] et de l'analyse dynamique[2] (automatisée ou non) de code. En effet, il a été observé que l'analyse statique manuelle et l'analyse dynamique sont toujours susceptibles d'omettre des vulnérabilités significatives, même sur du code de taille réduite.

Il s'avère nécessaire d'imposer le recours à l'*analyse statique outillée* afin de viser l'exhaustivité et la répétabilité de l'activité d'analyse de code par différents CESTI. L'ANSSI considère donc cette exigence comme une interprétation directe de la [CEM] au regard de l'état de l'art actuel.

Il est à noter que durant une évaluation, l'analyse manuelle peut être utilisée pour d'autres besoins, par exemple pour établir la traçabilité des SFR ou en guise de complément aux tests fonctionnels. Ces activités sont hors périmètre de cette note. Plus généralement, cette note ne touche à aucune autre famille d'exigences qu'AVA_VAN et ADV_IMP.

---

[1] par exemple une relecture critique de code.

[2] L'analyse dynamique de code requiert l'exécution du programme interprété ou compilé : il peut s'agir de tests fonctionnels ou de fuzzing par exemple.

2 Exigences pour le commanditaire

E1. Le commanditaire doit fournir l'intégralité du code source du produit à l'évaluateur et identifier toutes les parties de la TOE implémentées par des composants en source fermée.

Remarque : cette exigence est la transposition directe de [CC][3] pour l'activité ADV_IMP.1 : même si l'évaluateur n'est censé vérifier qu'un échantillon du code source il lui appartient de définir de façon impartiale et indépendante l'échantillon qu'il souhaite évaluer. De plus, l'exigence permet de disposer de l'intégralité du code source afin de pouvoir recompiler la cible d'évaluation correctement. L'évaluateur pourra refuser une fourniture incomplète en s'appuyant sur la section 3.5.4. Refus de fournitures de la [NOTE-20].

Remarque : La fourniture des bibliothèques en source ouverte n'est pas obligatoire, à condition que le commanditaire donne à l'évaluateur les directives pour en récupérer le code source (version exacte, lien *github*, …) et qu'il se soit engagé par écrit à ne pas avoir modifié ces librairies.

Remarque : pour AVA_VAN.3, le centre de certification pourra permettre à un commanditaire de donner l'accès à son code source dans ses locaux plutôt que de le livrer au CESTI. Cela ne peut être fait qu'au cas par cas, et demandera l'assurance que l'analyse faite par le CESTI n'est pas défavorablement impactée (le CESTI devra, par exemple, être capable d'installer ses outils d'analyse dans les locaux du développeur). Cette dérogation n'est pas possible pour les évaluations visant au-delà de AVA_VAN.3

Remarque : par définition, l'analyse du code est impossible pour les composants en source fermée. Il est donc crucial que ces composants soient tous identifiés.

E2. Le commanditaire doit fournir à l'évaluateur l'intégralité des éléments permettant la recompilation de la cible d'évaluation (outils, scripts…).

Remarque : cette exigence est la transposition directe de [CC] pour s'assurer de la complétude[4] et surtout de la bonne compréhension de l'implémentation par l'évaluateur.[5]

Remarque : cette exigence implique la fourniture de moyens de compilation propriétaires, lorsqu'ils sont nécessaires. « Compilation » est entendu ici dans le sens logiciel du terme.

EXEMPLE : pour un produit comme une passerelle de comptage communicant, la "compilation" sera la compilation du firmware qui sera ensuite chargée sur la TOE.

Avertissement : attention, si le commanditaire est tenu de fournir les moyens de construction de la TOE, il n'est pour autant pas tenu de les documenter (la documentation de ces outils n'est requise que si le composant ALC_TAT.1 est sélectionné). En revanche, le commanditaire doit assister l'évaluateur si ce dernier ne parvient pas à mettre en œuvre correctement la chaîne de compilation fournie. Ceci ne fait pas l'objet d'une exigence spécifique, car ce type de soutien est attendu plus généralement au titre de la [NOTE-20]. De la même manière, dans le cas où l'analyse statique outillée du code nécessite de « bouchonner[6] » certaines parties du code (par exemple des parties écrites

[3] Cf. Part 3 §13.3 "*The entire implementation representation is made available to ensure that analysis activities are not curtailed due to lack of information. This does not, however, imply that all of the representation is examined when the analysis activities are being performed.*".

[4] Cf. Part 3 ADV_IMP.1.1C "*The implementation representation shall define the TSF to a level of detail such that the TSF can be generated without further design decisions*".

[5] Cf. Part 3 §13.3 "*it is important that such "extra" information or related tools (scripts, compilers, etc.) be provided so that the implementation representation can be accurately determined*".

6 Un « bouchon » (ou *stub*) est une alternative temporaire à un code qui n'est pas utilisable directement par un autre code. Idéalement cette alternative doit avoir les mêmes caractéristiques fonctionnelles que le code remplacé. À défaut cette alternative peut correspondre à un code n'effectuant aucun traitement et retournant toujours le même résultat ou une version simplifiée du code initial traduisant le comportement du code remplacé. Le but de ce processus est ici de permettre de poursuivre l'analyse statique outillée même en présence de parties de code non analysables directement par l'outil.

dans un langage non géré par l'outil comme de l'assembleur), le commanditaire doit assister l'évaluateur pour assurer de la pertinence du bouchonnage.

3 Exigences pour l'évaluateur

E3. Si le composant d'assurance AVA_VAN.3 (ou supérieur) est sélectionné, l'évaluateur doit définir et utiliser une méthodologie d'analyse statique outillée du code respectant la table des matières donnée en Annexe A[7]. En particulier, cette méthodologie doit s'appuyer sur une analyse de code outillée, suivant une approche structurée. L'analyse manuelle ne doit être utilisée qu'en complément de cette analyse outillée, en particulier pour l'interprétation des résultats de l'outil et définir la criticité des défauts retournés (difficulté d'exploitation, criticité associée…) et pour effectuer les vérifications que les outils ne permettent pas de faire[8].

Remarque : La [CEM] n'impose pas le recours à une analyse statique outillée. Il s'agit ici d'un choix du centre de certification motivé par l'état de l'art.

Remarque : Le format de la méthodologie n'est pas imposé. Le CESTI est invité à réfléchir au support le plus approprié, afin de s'assurer de la maîtrise de la méthodologie par les évaluateurs. Il peut être ainsi préférable pour certains CESTI, par exemple, de définir leur méthodologie sous forme d'un support de formation, ou d'une aide en ligne pour leur outil d'analyse.

Remarque : La méthodologie sera validée dans le cadre de l'agrément du CESTI, en prenant en compte son application sur les projets d'évaluation.

E4. Si le composant d'assurance AVA_VAN.5 est sélectionné, l'analyse de code doit couvrir l'intégralité du code source.

Remarque : Cette exigence traduit le fait que le potentiel d'attaque retenu à AVA_VAN.5 inclut des scénarios d'attaques prévoyant des cotations à 24 points selon [CC], ce qui inclut en particulier des attaques par analyse exhaustive du code source :

a) *Elapsed Time* : un mois de charge (4) ;

b) *Expertise* : un expert en génie logiciel (7) ;

c) *Knowledge of TOE* : connaissance de l'intégralité du code source y compris les parties les plus critiques (11) ;

d) *Window of Opportunity* : attaques non sujettes à fenêtre d'opportunité (0) ;

e) *Equipment* : équipement logiciel spécialisé (2, voir [NOTE-18]).

E5. Si le composant d'assurance AVA_VAN.3 ou AVA_VAN.4 est sélectionné, l'évaluateur doit justifier son choix d'échantillonnage par les scénarios d'attaques envisagés au titre de cette activité d'évaluation.

Remarque : Cela implique en particulier que l'évaluateur ne peut pas se contenter d'analyser uniquement le code implémentant les fonctions de sécurité, en laissant de côté le reste : il s'agirait d'une grave incompréhension des principes de l'analyse de vulnérabilités, puisqu'une interface de la TOE n'implémentant aucune fonction de sécurité pourrait, par exemple, permettre une exécution de code arbitraire. À l'inverse, il est recommandé de fonder l'approche d'échantillonnage du code en priorisant les parties accessibles à l'attaquant (par exemple TSFI), puis à y ajouter d'autres parties au fur et à mesure de l'analyse en fonction des résultats obtenus.

---

[7] Cette annexe, destinée à être partagée dans le cadre de la normalisation ou d'accords de reconnaissance internationaux, est rédigée en anglais. Il est bien entendu possible pour le CESTI de définir une méthodologie en langue française.

[8] Par exemple :

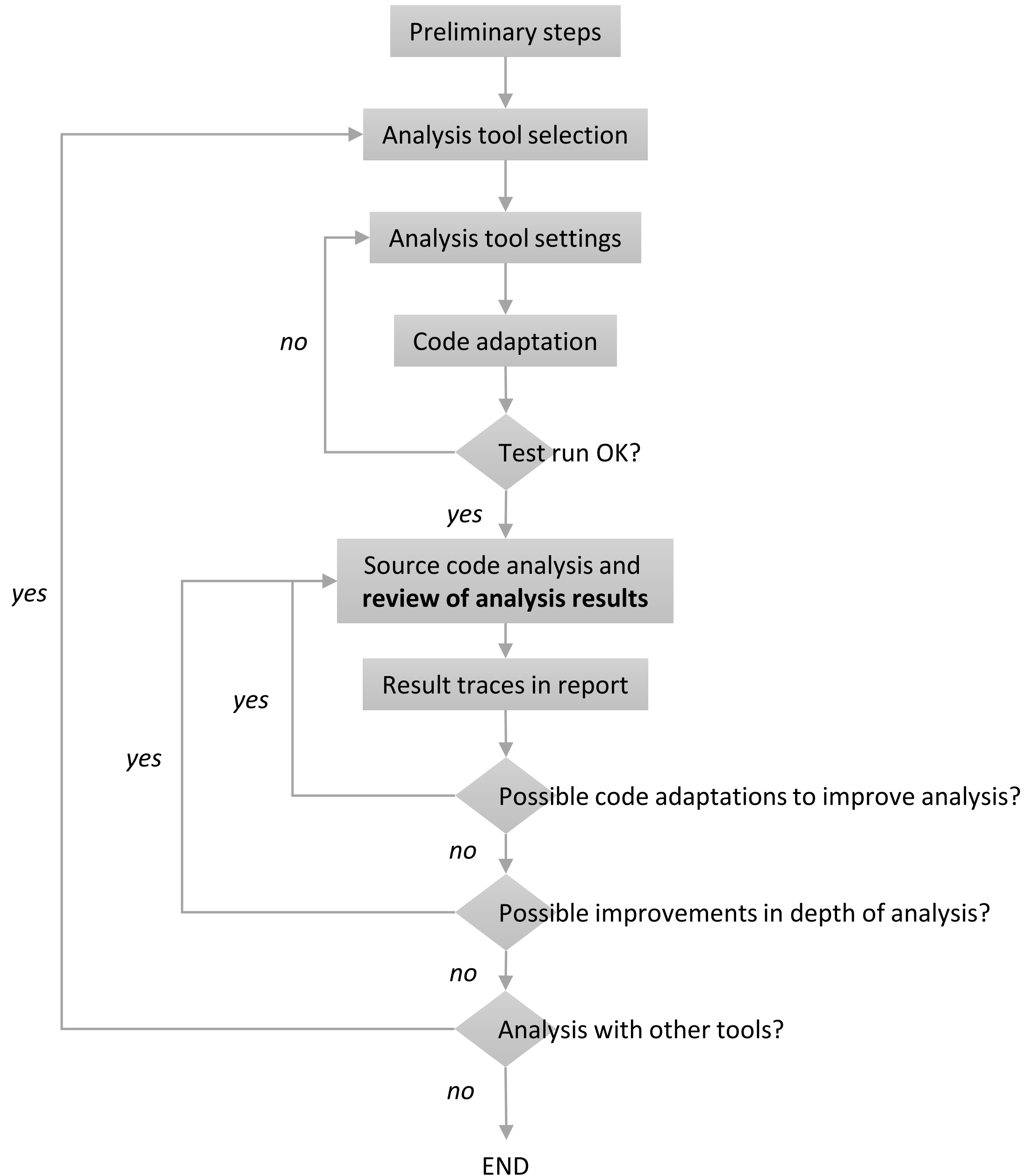
- vérifier l'implémentation correcte d'un algorithme ou d'un protocole (p.ex. respect d'une RFC ou algorithme cryptographique, etc.),
- vérifier la présence ou l'absence d'une vulnérabilité dont la sémantique n'est pas détectable par un outil seul (p.ex. vérification du masquage d'une donnée confidentielle, etc.),
- …

Remarque : L'analyse concerne aussi les composants en source ouverte utilisés par la TOE. Cela dit, le point d'entrée de l'analyse statique étant situé dans le code propre au développeur, l'outil devrait limiter de lui-même son analyse aux seules parties de code tierce partie utilisées par la TOE.

ANNEXE A. Table of contents for code analysis methods

1. Introduction

The main sections of this method correspond to the steps summarized in the figure hereafter. All steps are mandatory.



2. Preliminary steps

2.1. Inputs from the developer

The methodology shall describe how the evaluator recovers and manages the necessary inputs:

- *All source files of the TOE;*
- *All makefiles/scripts (or other project management files);*
- *Third party components (such as libraries).*

2.2. Analysis of the compilation

The same code gives different runtime behaviours depending on the compilation options: therefore, the evaluator must examine the code and the compilation process, so that their analysis is complete. Compilation options must also be considered while studying the exploitability of code vulnerabilities.

The methodology shall describe the following activities:

- *Clear identification of the tools used for the build:*
  - *Reference and version of the building tools (such as compilers),*
  - *Presence of proprietary tools (in that case the evaluator is mandated to ask the developer to provide the tool),*
  - *Hardware target(s);*
- *Verification that the compilation step is successful and can be repeated independently by the evaluator at their premises;*
- *Analysis of the compilation phase – typically check the makefiles to ascertain the use of:*
  - *Good practices,*

    EXAMPLE: some options can allow developers to detect errors at the compilation phase, such as the gcc option -Wall -Wextra. See also e.g. sections 5.1 et 5.2 of [Guide C]
  - *Hardening.*

    EXAMPLE: some options can help mitigate the exploitation of errors at runtime, such as the gcc option -Wformat. See also e.g. section 5.3 of [Guide C].

This phase may include back-and-forth exchanges between evaluator and developer: for large codebases, the evaluator may require that the delivered code already passes successfully a compilation with a restrictive set of options, so as to reduce the potential number of errors during the static analysis.

The methodology shall require the evaluator to provide the exact configuration for the build (e.g. complete list of options)

3. Analysis tool selection

3.1. List of static analysis tools used by the evaluator

The methodology shall select tools that detect code vulnerabilities (e.g. based on security coding guidelines as CERT, ISO 17961 or ANSSI guidelines, and on runtime-error detection).

NOTE:

- *Extended grep-like tools can be used but are not sufficient;*
- *Analysis tool oriented towards code quality and metrics are also welcome but do not address the needs of this methodology.*

The selected tool must have a code coverage feature to identify potential misuses[9] of the tool and to know precisely which parts of the code were analysed.

[9] An insufficient code coverage is a good indicator of a bad configuration of the tool.

3.2. Identification of constraints

The methodology shall define control points to check that the tool is appropriate with regard to the properties of the TOE, including at least the following:

- *The languages analysed by the tool cover the actual languages of the TOE (multi-language, possible presence of assembly code in the TOE…);*
- *The tool supports the hardware architecture of the TOE;*
- *The types of defects checked by the tool are consistent with the requirements of the evaluation (requirements may be specific to a TOE, but the methodology should select a set of requirements that will be used by default);*
- *The tool is capable to manage the code size and complexity of the TOE;*
- *The tool provides information on code coverage.*

3.3. Recommendations

As the evaluator is expected to perform an exhaustive analysis (within the defined scope) of the tool results, it is highly recommended to leverage any tool property that reduces the overhead during the analysis, notably:

- *Minimal code adaptation: The methodology should favour tools requiring minimal code adaptation.*
- *The use of a "sound" tool is only a good practice in presence of a compatible compiler/architecture/code size/code complexity: it is not mandatory;*

NOTE: "sound tools" are tools that can formally demonstrate the absence of a specific vulnerability. By design such tools require a high level of expertise.

- *Level of automation: automatic ability of the tool to manage the settings via Makefile or project manager.*

4. Analysis tool settings

The methodology shall address how the evaluator is supposed to setup the selected tools, in order to ensure the relevance of the analysis. This includes, but is not limited to:

- *Identification of all the needed source files (so as to make sure that the analysis can actually take place);*
- *Support of various compilers/architectures;*
- *Possibility to define some "implementation defined" aspects for « uncommon » targets;*
- *Representation of types: size, signedness, endianness…;*
- *Definition of entry points, initialization, interrupts and so on.*

5. Code adaptation for analysis

5.1. General principles

Adaptations should only be made if necessary to use the tool, especially since the code must stay representative of the evaluated TOE. However, the evaluator should not abandon the analysis simply because adaptations were needed.

5.2. Guidance for code adaptation

The methodology shall require the evaluator to justify the adaptations made to the code, and especially demonstrate that the adaptations themselves did not introduce any flaw detected later in the analysis and did not hide other issues.

The methodology shall describe how the evaluator is supposed to adapt the code to take care of specific constructions of the compiler/architecture (e.g. "implementation-defined" code) when it is not automatically handled by the tool or by a proper configuration of the tool.

The methodology shall describe the method used by the evaluator to manage multi-language code, notably the stubbing of parts not managed by the tool (e.g. assembly code or some parts of source code with a language not handled by the tool). Stubbing is not supposed to be used to handle missing parts of the code, since the full codebase is normally accessible to the evaluator.

6. First run/test step

The first test run is meant to check whether the tool can actually manage the code.

Consequently, the method shall describe how the evaluator:

- *selects low precision settings for the tool (minimum checks);*
- *analyses the first results, i.e.:*
  - *assesses informally the code quality,*
  - *analyses the code coverage.*

If this first run shows a bad configuration (the code coverage is often a good indicator), the methodology shall mandate the evaluator to return to the previous steps (code adaptation and/or tool settings), or even return to the developer (if the code quality is lacking and requires action from the developer).

When the coverage and the code quality are acceptable with regard to the method, the evaluator can start the real code analysis.

7. Real code analyses

In order to avoid being drowned in returned defects, the methodology shall mandatorily describe a step-by-step process when the tool allows it.

- *First step: the evaluator will ultimately try to check the adherence of the code to a large set of requirements. However, in a first step, the evaluator should select a first, smaller, subset of classic flaws. As a priority, these checks should target undefined or unspecified behaviours and classic vulnerabilities.*

EXAMPLE: if the evaluator uses CERT-C as a set of requirements, the first run could focus on the Level 1 rules (High severity, likely, inexpensive to repair).

- *A first review of tool results (see following section) will be performed.*
- *As a second step, the evaluator will typically increase the tool precision in a new run and perform a new review of tool results.*

EXAMPLE: if the evaluator uses CERT-C this would typically consist in adding Level 2 and Level 3 rules.

It is better, when possible by the tool, to organize the analysis in several runs, corresponding to increasingly "aggressive" settings of the tool. In all cases, a rigorous analysis of the code is expected and a single run with a low level of aggressiveness is not sufficient.

If there is remaining time after the last step of code analysis, the evaluator can activate more specific checkers in a new run to detect more complex vulnerabilities or to find issues related to code quality.

EXAMPLE: it means focusing on more complex vulnerabilities (after checking buffer overflows, the evaluator now checks TOCTOU vulnerabilities) or to look for dangerous features in the code.

8. Review of tool results

8.1. General method and interpretation guidelines

*The review of results is a **critical and non-accessory step**: tools outputs shall never be copy-pasted in an ETR without an actual expert opinion on the findings. The evaluator is expected to have the critical distance to the results provided by the tool.*

The methodology shall therefore define how this review is performed, including, but not limited to:

- *the definition of priority classes for returned defects;*
- *the method used to interpret unreachable code (initialization issue, bad settings, defensive parts of code, interrupts…), so that the evaluator is able to return to previous steps and fix the tool settings and/or adapt the code accordingly. Unreachable code can be indicative of an error in code adaptation or/and in tool settings.*

The methodology shall provide the evaluator with clear guidelines to:

- *take care of false positives (i.e. clearly identify them in the tool if possible, and keep a trace of false positives in the report);*
    - *As per [NOTE-20], the evaluator is allowed to reject the TOE if the number of findings exceeds its analysis capacity: if the evaluator considers the code as not mature enough for analysis, they may provide the list of findings to the developer and ask the developer to fix the code and/or provide a rationale that these findings are false positives;*
    - *However, the evaluator still has the final responsibility of qualify which warning are false or true positives, and will ultimately have to challenge the developer information.*
- *decide when a new tool run is needed, and with which additional checks if the remaining evaluation time allows it;*
- *decide when another tool shall be used (return to tool selection) if the remaining evaluation time allows it.*

The methodology shall provide guidance to rate the identified vulnerabilities according to the rating system of the evaluation method. In particular, even if the vulnerability was found by code analysis, the evaluator shall consider the possibility that the same vulnerability is found by other means. The final rating shall be the smaller rating of all considered scenarios.

The methodology shall include guidelines and/or pre-ratings for the following notions:

- *Elapsed Time: time taken to identify and exploit specific classes of vulnerabilities, e.g. by comparison to challenges or CTF exhibiting similar vulnerabilities;*
- *Level of expertise:*
    - *If the attack scenario supposes that the attacker has the knowledge of the source code, only a single expert should be required at most;*
    - *If the attack scenario supposes that the attacker can discover the vulnerability in a black box setting, additional expertise may be required (Hardware expertise to access the TOE, network expertise to perform fuzzing, etc.);*
- *Knowledge of the TOE:*

  - *This rating may be low or zero in scenarios where the attacker discovers the vulnerability in a black box setting, or if the vulnerability is in an open source component;*
  - *The rating should reflect the protections in place to protect the source code, which may require a developer interview: if the evaluation requires ALC_DVS.2, access to source code may be rated up to 11 points. If the developer cannot show strong access control, the rating may be as low as 3 points;*
  - *The rating should also consider the possibility for an attacker to obtain a binary, and reverse engineer it to find vulnerabilities;*
- *Window of opportunity;*
- *IT hardware/software or other equipment required for exploitation: as per [NOTE-18], the rating cannot be higher than 2 for commercial tools.*

As a general rule, it is not recommended to try and rate all issues identified by the tool, as it would result in a very time-consuming process. Quite the contrary, the evaluator should instead try to define a general rating for a given class of vulnerabilities (for example use-after-free) and apply it to every violation of this class, unless it is a false positive.

8.2. About the possible fixes to improve analysis

When reviewing analysis results, evaluators and developer face a practical issue: some vulnerabilities may cause the tool to stop its analysis, leaving a large part of the unanalysed code. Some tools, when finding an error or potential error, may resume the analysis by excluding the associated erroneous context(s) of this (potential) error, leaving also a part of the code unanalysed.

EXAMPLE: loop termination errors or always true/false conditions resulting in unreachable code that was not supposed to be unreachable.

The methodology shall describe:

- *in which cases the evaluator should try to fix the bugs identified in the code (or ask the developer to fix them), or when they should ask clarifications to the dev;*
- *how the evaluator is supposed to trace these cases in the ETR.*

The evaluator may simply ask for the developer to provide a fixed source code, and resume their analysis. However, other blocking errors may be found again, leading to a possibly large number of iterations between the evaluator and the developer. Alternatively, the evaluator may perform a small adaptation of the code, so that the analysis can resume. This offers the developer a more complete set of findings in a single evaluation phase. In any case, the evaluator should synchronize the further procedure with the developer before adjusting the code.

EXAMPLE: EASY/OBVIOUS FIX – use of an unsigned int counter instead of signed one resulting in an infinite loop: the evaluator changes the unsigned counter to a signed one, only to allow the analysis to resume.

EXAMPLE: the evaluator erases the whole loop that presents a termination error.

WARNING: as clearly highlighted by the previous examples, such adaptations are neither fixes nor patches, and may result in non-functional code. They are only meant to allow the analysis to resume, and should never been reintroduced in the product code as is.

ANNEXE B. Références

Référence	Document
[NOTE-18]	ANSSI-CC-NOTE-18 - Prise en compte des outils dans les évaluations logicielles
[NOTE-20]	ANSSI-CC-NOTE-20 - Règles relatives à la mise en œuvre des évaluations sécuritaires
[CC]	*Common Criteria for Information Technology Security Evaluation:* *Part 1 : Introduction and general model, avril 2017, version 3.1, révision 5, référence CCMB-2017-04-001* ; *Part 2 : Security functional components, avril 2017, version 3.1, révision 5, référence CCMB-2017-04-002* ; *Part 3 : Security assurance components, avril 2017, version 3.1, révision 5, référence CCMB-2017-04-003*.
[CEM]	*Common Methodology for Information Technology Security Evaluation - Evaluation methodology, April 2017,* version *3.1 révision 5,* référence CCMB-2017-04-004.
[Guide C]	*ANSSI - guide - Règles de programmation pour le développement sécurisé de logiciels en langage C - v1.4 du 24/03/2022*.

La plupart de ces documents peuvent être consultés et téléchargés depuis le site de l'ANSSI (www.ssi.gouv.fr).