**Université du Québec en Outaouais**

**Département d’informatique et d’ingénierie**

**INF 4173 - Projet synthèse**

**Rapport final**

Analyse des vulnérabilités cryptographiques catégorisées par des logiciels d'analyse

**Par**

**Jérémy Bolduc - BOLJ30039900**

**Jason Lafrenière Nickopoulos – LAFJ09029509**

**Présenté à**

**Raphaël Khoury  - Superviseur**

**M. Karim El Guemhioui - Coordonnateur**

**19 Décembre 2022**

**Table des matières**

1. Introduction
2. Méthodologie

2.1. Catégorisation des vulnérabilités

2.2. Les listes CVE et CWE

2.3. sources des vulnérabilités cryptographiques

1. Objectifs
2. Obstacles

4.1 Travaux connexes

          4.2 Complexité de l’analyse des vulnérabilités

4.3 Sélection des outils d’analyse

4.4 Trie de toutes les fichiers entre les versions vulnérables et    réparées

4.5 Construction de la base de données

1. Taxonomie des *CVE*
2. Outils d’analyse

6.1. *Bandit* : *Python*

6.2. *Visual Code Grepper* : *C*, *C++*, *Java*, *PHP*

6.3. *DeepScan* : *JavaScript*, *TypeScript*

6.4. *Staticcheck* : *Go*

1. Résultats des outils d’analyses

7.1 Analyse du code vulnérable

7.2 Analyse du code réparer

1. Analyse des résultats

8.1 Informations primaires

8.1.1 Versions défaillantes

8.1.2 Versions Corrigées

8.2 Informations secondaires

8.2.1 Versions défaillantes

8.2.2 Versions Corrigées

1. Conclusion
2. Bibliographie

**Table des figures**

Figure 1. Exemple de recherche possible sur CES détails……………………………………..…9

Figure 2. Exemple de vues possible dans la liste CWE………………………………….………11

Figure 3. Pourcentage des langages de programmation par catégories de vulnérabilités cryptographiques……………………………………………………………………………...….14

Figure 4. Résultats d’analyse du code vulnérable du CVE-2022-1235…………………….…....20

Figure 5. Résultats d’analyse de la version défaillante de la CVE-2019-11578………………...21

Figure 6. Résultats d’analyse de la version défaillante de CVE-2020-8867..…………………...21

Figure 7. Résultats d’analyse de la version défaillante de CVE-2020-12735…………………...22

# **Résumé**

Le développement de code peut être un processus long et ardu. Des dizaines voir, des centaines de fonctions interagissent les unes avec les autres et les développeurs doivent s’assurer que tout est bien orchestré afin de pouvoir livrer un produit efficace et compétent. Cependant, lorsque le code est à caractère cryptographique, cette orchestration est encore plus importante. Certaines fonctions ou méthodes d’implémentation peuvent exposer des faiblesses et rendre des informations sensibles à risque si mal faites. Dans la littérature, ces faiblesses s’appellent des vulnérabilités. Une fois détectées, elles peuvent être mises dans une liste, appelée *Common Vulnerabilities and Exposure* ou *CVE*, cette liste aide les professionnels des technologies de l'information à coordonner leurs efforts afin de corriger les erreurs et rendre leurs systèmes plus sécuritaires.

Des outils permettant de détecter les fautes de programmation existent aussi afin d’aider les développeurs dans leur processus de création. De la sorte, ils peuvent analyser le code et prévenir les erreurs cryptographiques qui peuvent se retrouver dans leur code avant qu’il soit déployé et risque d’exposer des informations critiques.

Ainsi, l’objectif du projet est d’utiliser des outils d’analyse de code sur des *CVE* connues avant et après qu’une correction ait étée implémentée. De la sorte, nous voulons voir si ces outils sont capables de détecter des fautes cryptographiques présentes dans ces projets et de voir comment ils réagissent après qu’une correction soit implémentée.

Parmi les 30 *CVE* analysées, seulement une des trente a produit une détection valide de la vulnérabilité (vrai positif). Il s’agit d’un taux de succès de 3.33%. C’est très bas comme taux de succès et c’est résultats sont moins que l’on pensait avoir comme taux de succès. La majorité des *CVE* que nous avons répertoriées comprenaient des problèmes qui se situent dans la signature d’une fonction. C’est quelque chose que les outils que nous avons utilisés devraient être capable de détecter car ils utilisent des méthodes d’analyse statique.

Bien que le taux de succès soit faible, cela ne veut pas dire que les outils n’ont rien produit. Plusieurs avertissements ont été émis pour les *CVE* que nous avons échantillonnés. Ces avertissements n’étaient juste pas ceux que nous recherchions.

Ainsi, on peut tirer de ces résultats quelques recommandations:

* Les outils pourraient bénéficier d’ajout de règles d’analyse de certaines fonctions pour être en mesure d’analyser les signatures de fonctions en lien avec les *CVE*
* Le développement d’un programme permettant la diminution d’effort manuel pour répertorier, analyser et acquérir les données en lien avec les *CVE* et *CWE*

# **Introduction**

L’introduction de l’informatique a été une étape très importante dans le développement du monde contemporain. Initialement, elle était utilisée à des fins très spécifiques et dans des domaines spécialisées. Avec le temps, elle a été intégrée dans la société et fait maintenant partie de la vie quotidienne de tout le monde. Elle enrichit et facilite le développement de pratiquement tous les domaines, soit directement (l’utilisation des ordinateurs en finance) ou indirectement (par communication de téléphone cellulaire entre des individus). Les bénéfices de son utilisation augmentent à chaque année mais elle ne sont pas sans risque. Les données transmises à travers des outils informatiques peuvent être très importantes et en sorte doivent rester cachées par les yeux des personnes non concernées. Pour réaliser cette tâche ardue et complexe, la cryptographie est utilisée. L’importance de la cryptographie dans le cybermonde est non questionnable, elle doit être efficace car les conséquences d’une cryptographie faible peuvent être dévastatrices. La fondation à but non lucratif *OWASP* (*Open Web Application Security Project*) considère les vulnérabilités cryptographiques comme étant la deuxième catégorie de vulnérabilités les plus importantes dans l’industrie de la cyber sécurité présentement [12].

Comme n’importe quel mécanisme de sécurité, il va toujours y avoir des acteurs malicieux voulant briser cette sécurité pour avoir accès aux ressources protégées. L’implémentation de ces mécanismes de sécurité peut être très complexe dans un système informatique, ceci est doublement vrai pour la cryptographie. Pour faciliter l’utilisation de la cryptographie, plusieurs librairies existent à la portée des développeurs de système informatique. Ces librairies contiennent des méthodes nécessaires pour l’utilisation de la cryptographie comme le cryptage, le décryptage, l’authentification et la validation. Par contre, ces librairies doivent constamment être en évolution pour répondre au besoin de sécurité nécessaire à la protection des données informatiques. Les changements sont faits par des experts dans le domaine (ou du moins des personnes qui sont très familières avec les concepts cryptographiques). Cette évolution rapide peut causer des vulnérabilités et faire en sorte que les données ne soient pas aussi bien protégées qu’on le pense. Malheureusement, ceci est seulement une de plusieurs façons qu’une vulnérabilité cryptographique peut se manifester. Maîtriser le processus de cryptographie est seulement une composante de l’enjeux. Il faut aussi maîtriser les éléments qui sont manipulés, plus spécifiquement la mémoire et les structures de données. Pour ce qui concerne les vulnérabilités provenant des librairies cryptographiques, [22] disent que seulement 17% des vulnérabilités provenaient du code source. Les vrais problèmes proviennent de la façon dont les développeurs utilisent ces librairies.

Pour aider les développeurs à minimiser les erreurs produites lors du développement d’un produit, plusieurs outils peuvent être utilisés. Ces outils peuvent être spécialisés pour identifier des fautes de mémoire, des fautes de cryptographie ou autres. Pour trouver ces fautes, il existe plusieurs méthodes d’analyse que ceux-ci peuvent exploiter. Un bref aperçue de ces méthodes :

* [23] *Static Application Security Testing* (*SAST*) : ne voit pas les aspects d'exécution (*white box security testing*). L’application est testée de l’intérieur vers l’extérieur. Cette méthode nécessite le code source.
* [24] *Dynamic Application Security Testing* (*DAST*) : les outils testent le code en cours d'exécution (*black box security testing*). Ne voit la cryptographie que de l’extérieur de l’application. Les types de comportement qui ne peuvent être observés qu'au moment de l'exécution :
  + les valeurs de clé chargées à partir des magasins de clés
  + mots de passe dans les fichiers de configuration
  + générateurs de nombres aléatoires…, etc.
* [25] *Interactive Application Security Testing* (*IAST*) : Tests interactifs de sécurité des applications. Cette méthode est utilisée pour tester tout ce qui est spécifiquement exercé par les tests fonctionnels.

La méthode la plus simple est l’analyse statique. Dans le cadre du projet, ce sont des outils utilisant cette méthode d’analyse que nous allons utiliser. Ceci est principalement à cause de l’aspect *open source*, les outils utilisant les autres méthodes sont très dispendieux, par exemple *Cryptosense* [17].

# **Méthodologie**

## **Catégorisation des vulnérabilités**

Les catégories des types de vulnérabilités cryptographiques utilisées dans ce projet sont celles définies dans le *framework Tafelsalz* [1]. Ces catégories sont :

1. *Initialization*
   1. *Predictable Sequences*
   2. *Re-use*
   3. *Weak Values*
   4. *Source*
2. *Insecure Defaults*
3. *Weak Algorithms*
4. *Validation*
5. *Persistence of Secrets*
6. *Usage Complexity*
7. *Knowledge Base*
8. *Identity Management*
9. *Password Hashing*
10. *Other*

# **Les listes *CVE* et *CWE***

Dans la littérature, une des ressources les plus importantes utilisées pour identifier les types de faiblesses logiciels ainsi que matériel liées à la cyber sécurité est la liste *CWE* (*Common Weakness Enumeration*) [5]. Cette liste contient 933 vulnérabilités à la date de dépôt de ce rapport. Les étiquettes *CWE* sont liées spécifiquement à des vulnérabilités *CVE* (*Common Vulnerabilities and Exposures*). Lorsqu’une vulnérabilité est découverte, elle est associée à une ou plusieurs versions d’un produit du fabricant respectif. Elle peut aussi être associée à une librairie et citer les produits ainsi que les versions affectées par cette librairie. Ensuite, un identifiant *CWE* peut être attribué pour expliquer le problème causé par ce type de vulnérabilités. Avant l’année 2019, une bonne partie des *CVE* en lien avec les vulnérabilités cryptographiques étaient associées à l’étiquette 310. Après 2019, la liste a été modifiée pour inclure des types de vulnérabilités plus spécifiques pour avoir un meilleur classement de ceux-ci. Le mappage entre les *CVE* et *CWE* se fait manuellement par des spécialistes du domaine [8].

Un excellent point de référence qui nous a aidé à trouver les *CWE* en lien avec la cryptographie sont ceux qui ont été mappées par la fondation *OWASP*. La liste qui suit comprend tous les *CWE* que nous avons étudiés dans notre projet dans le but de mieux comprendre les types de vulnérabilités liées à la cryptographie.

1. *CWE-261 Weak Encoding for Password*
2. *CWE-296 Improper Following of a Certificate's Chain of Trust*
3. *CWE-310 Cryptographic Issues*
4. *CWE-319 Cleartext Transmission of Sensitive Information*
5. *CWE-321 Use of Hard-coded Cryptographic Key*
6. *CWE-322 Key Exchange without Entity Authentication*
7. *CWE-323 Reusing a Nonce, Key Pair in Encryption*
8. *CWE-324 Use of a Key Past its Expiration Date*
9. *CWE-325 Missing Required Cryptographic Step*
10. *CWE-326 Inadequate Encryption Strength*
11. *CWE-327 Use of a Broken or Risky Cryptographic Algorithm*
12. *CWE-328 Reversible One-Way Hash*
13. *CWE-329 Not Using a Random IV with CBC Mode*
14. *CWE-330 Use of Insufficiently Random Values*
15. *CWE-331 Insufficient Entropy*
16. *CWE-334 Small Space of Random Values*
17. *CWE-335 Incorrect Usage of Seeds in Pseudo-Random Number Generator(PRNG)*
18. *CWE-336 Same Seed in Pseudo-Random Number Generator (PRNG)*
19. *CWE-337 Predictable Seed in Pseudo-Random Number Generator (PRNG)*
20. *CWE-338 Use of Cryptographically Weak Pseudo-Random Number Generator(PRNG)*
21. *CWE-340 Generation of Predictable Numbers or Identifiers*
22. *CWE-347 Improper Verification of Cryptographic Signature*
23. *CWE-523 Unprotected Transport of Credentials*
24. *CWE-757 Selection of Less-Secure Algorithm During Negotiation('Algorithm Downgrade')*
25. *CWE-759 Use of a One-Way Hash without a Salt*
26. *CWE-760 Use of a One-Way Hash with a Predictable Salt*
27. *CWE-780 Use of RSA Algorithm without OAEP*
28. *CWE-818 Insufficient Transport Layer Protection*
29. *CWE-916 Use of Password Hash with Insufficient Computational Effort*
30. *CWE-1240 Use of a Cryptographic Primitive with a Risky Implementation*

## **Sources de vulnérabilités cryptographiques**

Pour être en mesure de trouver les données importantes à notre recherche, nous avons utilisé plusieurs méthodes, comme par exemple l’extraction des données des bases de données *open source*, comme par exemple *vulncode-DB* [3]. Elle contenait le code source vulnérable ainsi que réparé. Par contre, son utilité était limitée car ses méthodes de filtrage étaient très primitives. De plus, elle n’était pas à jour et pour certains *CVE* et des fois il manquait du code source qui était important à l’identification de la vulnérabilité. D’après notre recherche, la raison est simple, lorsqu’une vulnérabilité est patché par les développeurs concernées, c’est très rare que le patch contient seulement le changement pour en lien avec la *CVE*. Souvent, dans le cas des grandes entreprises corporatives, dans une journée il peut y avoir plusieurs *CVE* qui sont attribuées à leurs produits, en conséquence, ils sortent un grand patch pour réparer toutes ces vulnérabilités en même temps.

À cause de ces limitations, nous avons dû trouver d’autres moyens pour trouver le code source vulnérable associée aux *CVE* en lien avec la cryptographie. Plus précisément, nous avons utilisé *CVEdetails* [13], ce site comprend un moteur de recherche très sophistiqué. La *figure 1* permet de montrer un visuel des informations qui peuvent être utilisées pour trouver une *CVE* spécifique (ou tous les *CVE* en lien avec une étiquette *CWE*). Il est un site parmi plusieurs qui a été utile à notre recherche [2, 3, 4, 5, 7, 14]. Dans pratiquement tous les sites comme *CVEdetails*, il existe une section « *References* » qui contient des liens à d’autres sites qui aident à expliquer la problématique. C’est spécifiquement cette section qui nous a permis de trouver les ressources nécessaires pour l’analyse des *CVE*.

Malgré l’utilisation de ces méthodes, développer une façon automatique d’acquérir ces données n’a pas été réalisable pour la majorité des *CVE*. Nous avons été en mesure d’acquérir 11 *CVE* à travers un code python qui a été développé par un élève de notre superviseur. Ce code en fait retrouvait les *CVE* qui était située dans les bases de données *vulncode-DB* et *CVEfixes* [6]. Comme mentionné plus tôt, il manquait du code source important pour certains *CVE* alors il a fallu aller chercher le code source à partir de *CVEdetails*. Le programme utilisée :



Pour l’autre partie des *CVE*, nous avons parcourues les liens dans les sections « *References* » de chaque *CVE* pour trouver le *commit* spécifique qui comprenait les 2 versions de code recherchées (vulnérable et réparer). Souvent, ces liens étaient dirigés sur *GitHub* [29].

Graphical user interface, text, application, email

Description automatically generated

*Figure 1. Exemple de recherche possible sur CVEdetails*

# **Objectifs**

Le projet consiste à répertorier et analyser 30 *CVE* en utilisant les catégories énumérées par le *framework Tafelsalz*. Par la suite, des outils d’analyse cryptographique statique seront utilisés pour analyser le code source vulnérable ainsi que le code source réparer. Cela permet alors de tester les outils pour savoir s’ils sont en mesure d’identifier l’erreur appropriée dans le code (vrai positif), mais aussi de savoir s’ils sont capables de détecter le code réparer (vrai négatif suite à un vrai positif) et ne plus donner un avertissement de détection. Pour faciliter la tâche d’analyser le code source, nous avons pris les versions de code vulnérable avant et après le *commit* qui a réparé la vulnérabilité pour chaque *CVE*.

Par la suite, une analyse sera effectuée sur les résultats générés par les outils pour être en mesure d’évaluer leur performance. Pour ce faire, nous allons évaluer leur taux de succès (vrai positif) et leur taux d’erreur (faux négatif). De plus, nous allons énumérer les types de vulnérabilités qui ont été détectables par les outils d’analyse.

# **Obstacles**

## **Travaux connexes**

Initialement, nous croyons que l’obtention des données aurait été une tâche relativement facile avec l’utilisation des bases de données. Ceci n’a pas été le cas. Comme décrit dans le plan de travail, nous voulions développer un programme qui allait chercher les vulnérabilités dans une base de données pour ensuite les stockées dans un répertoire ou une base de données personnalisé. Ceci a fonctionné pour certains *CVE* (celles retrouvées dans *vulncode-DB*), mais pas pour la majorité des *CVE*. Même avec ces données, il a fallu aller chercher le code source de certains *CVE* car il manquait des données importantes pour notre recherche.

Dans le but de trouver une façon d’automatiser le processus, une recherche extensive a été faite. Lors de cette recherche, nous sommes tombés sur plusieurs articles qui ont démontré le même problème que nous avons eu à cet égard [15, 21]. La complexité liée à l’indexation des *CVE* et *CWE* rend le processus d’automatisation d'acquisitions des données très difficile. Les références qui sont associées aux *CVE* ne sont pas nécessairement utiles à l’acquisition du code source, pour la majorité des *CVE*, il existe seulement des liens à des rapports générés par les entreprise concernés (car le code source est privé).

Lors de notre recherche, on a réalisé que la méthodologie que nous avions développée était la même que celles utilisées par d’autres chercheurs [21]. Ils ont eux aussi cherché spécifiquement les vulnérabilités cryptographiques, par contre ils se sont limités à l’analyse des *CVE* en rapport avec les langages de programmation *C* et *C++*. Notre seule limitation était l’accessibilité au code source vulnérable ainsi qu’au code source réparer. Cette limitation  n’est pas négligeable car plusieurs *CVE* n’ont pas de solution (produit discontinué, abandonné, etc.). En moyenne, les vulnérabilités prennent entre 60 et 150 jours pour être réparées [27]. Ceci peut varier dépendamment de la complexité de la vulnérabilité et des ressources dépensées pour résoudre cette problématique.

## **Complexité de l’analyse des vulnérabilités**

La liste *CWE* peut être visualisée comme étant une structure arborescente profonde. Il existe plusieurs « vues » possibles pour filtrer les *CWE* en lien avec les intérêts de l’utilisateur. Un bref aperçu d’une infirme partie des possibilités de vues sont démontrés par la *figure 2* [28]. La structure complexe de la liste *CWE* cause de l’ambiguïté car plusieurs sous-catégories existent pour certaines catégories et ceux-ci se chevauchent entre différentes hiérarchies. Cette ambiguïté est aussi prononcée par le fait que la liste change avec le temps, et certaines catégories deviennent non utilisées, comme par exemple *CWE* 310, seulement les sous catégories de celles-ci sont utilisées après 2019.

         Pour être en mesure d'associer les *CVE* aux bonne catégories énumérées par le *framework Tafelsalz*, nous avons dû mener une analyse profonde sur

chaque *CVE* pour mieux comprendre la cause de ces vulnérabilités. Cette tâche en

soit a pris beaucoup de temps, les facteurs qui ont contribuées au temps d’analyse :

* La documentation disponible sur le problème et le produit.
* L’inclusion des autres changements qui n’était pas en lien avec la *CVE* entre les versions vulnérables et réparées.
* Le nombre de fichiers affectés entre les versions vulnérables et réparées.
* Le nombre de lignes de codes affectées dans chaque fichier entre les versions vulnérables et réparées.
* La complexité associée à la compréhension de la vulnérabilité.
  + Type de vulnérabilité logiciel (ex : librairie math.*random*() ou *crypto.random*()).
  + Type de vulnérabilité matériel (ex : architecture *MIPS*).
* Le langage de programmation utilisé.
  + Langage de haut niveau (ex : *Java*, *Python*, *Go*, …)
  + Langage de bas niveau (ex : *C*, *C++*, *Assembleur*, …)
* Les techniques de cryptographie utilisées.
  + *RSA*, *Elliptic Curve Diffie-Hellman* (*ECDH*), etc.

Text

Description automatically generated with medium confidence  
*Figure 2. Exemple de vues possible dans la liste CWE [28].*

## **Sélection des outils d’analyse**

Au début de la conception du projet, nous voulions utiliser seulement deux outils d’analyse pour ensuite comparer les résultats entre les deux. Le problème est que la plupart des outils sont spécialisés dans un sous-ensemble de langages de programmation. Dans notre analyse, nous avons considéré tous les *CVE* en lien avec la cryptographie, peu importe le langage de programmation utilisé. En conséquence, il a fallu plus que deux outils pour être en mesure d’évaluer chaque vulnérabilité trouvée. Pour cette raison, nous avons opté pour l’évaluation des résultats des outils d’analyse pour évaluer leur performance individuelle à la place de comparer la performance entre eux. Les langages de programmation utilisées dans les *CVE* sont :

* *C* = 10
* *JavaScript* = 5 (pour la *CVE-2021-32738* , c’est spécifiquement écrit en *TypeScript*)
* *Python* = 4
* *PHP* = 3
* *Go* = 3
* *Java* = 3
* *C++* = 2
* *Erlang* = 1 (inclus avec *C* dans la *CVE-2011-0766* car les deux langages sont impliqués)

## **Trie de toutes les fichiers entre les versions vulnérables et réparées**

Le volume de code impliquer dans certains *CVE* est étourdissant. Avant de faire le tri des fichiers, nous avions passé le code source directement dans les outils d’analyse, les résultats générer pouvaient dépasser 10 MB de données pour seulement le code source vulnérable ou réparer d’une *CVE*. Sans une analyse approfondie du code source, nous ne pouvions pas savoir où se situaient les vulnérabilités et si les outils pouvaient même les détecter. C’est principalement pour cette raison que nous avions fait un tri de chaque fichier incluent dans les réparation des *CVE*. Par exemple, il avait 11 fichiers affectés par les changements du *commit* qui réparait le code vulnérable du *CVE-2011-0766*. À la fin du trie, il restait seulement 3 fichiers. Ceci a non seulement aidé à l’analyse des vulnérabilités mais aussi l’analyse des résultats des outils d’analyse. Certains fichiers trier comprenait jusqu’à 7500 lignes de code.

## **Construction de la base de données**

Parmi nos objectifs initiales, nous voulions construire une base de données pour tester le code source ainsi que centraliser et organiser ces données. Lorsque le code source a été obtenu parmi les méthodes décrites précédemment, ils ont été catégorisés dans un répertoire commun. Cette étape s’est produite après la catégorisation des *CVE*, alors nous commençons par créer des répertoire parents qui était nommée par leur catégorie de vulnérabilités. Ajouter à la fin du nom est un chiffre entre parenthèses qui représente le nombre de *CVE* associés à ce type de vulnérabilité (par exemple, *Insecure Defaults*(1)).

Nous avons considéré la possibilité d’utiliser *SQLite* comme base de données, par contre après avoir finalisé l’analyse les *CVE*, nous avons conclu que ceci n’était pas idéal. Nous avons opté à la place de déposer toute la recherche sur *GitHub* avec une visibilité publique. Cette solution est beaucoup plus accessible et simple à réaliser et elle permet à la communauté d’utiliser les résultats de notre recherche à leurs propres fins.

* <https://github.com/GreekProgrammingGod/ProjetSynthese>

# **Taxonomie des *CVE***

* ***Initialisation (16)***
  + ***Predictable sequence (12)***
    - *CVE-2011-0766 : Erlang, C*
    - *CVE-2012-2417 : Python*
    - *CVE-2013-1445 : Python*
    - *CVE-2014-5386 : C++*
    - *CVE-2015-8867 : C*
    - *CVE-2018-12520 : C++*
    - *CVE-2019-11808 : Java*
    - *CVE-2020-12735 : Php*
    - *CVE-2020-28924 : Go*
    - *CVE-2021-3538 : Go*
    - *CVE-2021-41117 : JavaScript*
    - *CVE-2022-36045 : JavaScript*
  + ***Re-use (2)***
    - *CVE-2019-15075 : PHP*
    - *CVE-2022-1434 : C*
  + ***Weak Values (2)***
    - *CVE-2019-10908 : Java*
    - *CVE-2022-1235 : PHP*
* ***Insecure Defaults (2)***
  + *CVE-2012-3458 : Python*
  + *CVE-2016-1000352&1000344 : Java*
    - Ici, les deux *CVE* ont été combinés car c’est la même vulnérabilité qui cause la problématique pour les deux *CVE*. En d’autres mots, le patch pour un à régler le problème pour les deux.
* ***Other (4)***
  + *CVE-2013-2548 : C*
  + *CVE-2014-3570 : C*
  + *CVE-2014-8275 : C*
  + *CVE-2016-10530 : JavaScript*
* ***Usage complexity (5)***
  + *CVE-2017-7526 : C*
  + *CVE-2018-16870 : C*
  + *CVE-2018-19653 : Go*
  + *CVE-2019-9155 : JavaScript*
  + *CVE-2020-26263 : Python*
* ***Validation (3)***
  + *CVE-2016-2053 : C*
  + *CVE-2019-11578 : C*
  + *CVE-2021-32738 : TypeScript (JavaScript)*

Une description détaillée de la problématique est disponible pour chaque *CVE* dans le répertoire *GitHub* mentionnée précédemment. Le document nommé *Analyse\_CVE* à la racine du répertoire est là pour une vue rapide sur les fichiers et fonctions impliquées dans la vulnérabilité respective.

Pour ne pas avoir un biais sur une catégorie de vulnérabilité spécifique, nous avions variées quelques paramètres :

* Trouver des *CVE* à partir de différentes étiquettes *CWE* .
* Trouver des *CVE* se produisant à des années différentes.
* Trouver des *CVE* écrites en différents langage de programmation

De cette façon, nous avons été en mesure de quantifier un échantillonnage des vulnérabilités cryptographiques à travers les 11 dernières années utilisant langages de programmation variés.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ***C*** | ***JavaScript*** | ***Python*** | ***PHP*** | ***Go*** | ***Java*** | ***C++*** | ***Erlang*** |
| ***Initialisation*** | 30% | 40% | 50% | 100% | 67% | 67% | 100% | 100% |
| ***Insecure Defaults*** |  |  | 25% |  |  | 33% |  |  |
| ***Other*** | 30% | 20% |  |  |  |  |  |  |
| ***Usage Complexity*** | 20% | 20% | 25% |  | 33% |  |  |  |
| ***Validation*** | 20% | 20% |  |  |  |  |  |  |

*Figure 3. Pourcentage des langages de programmation par catégories de vulnérabilités cryptographiques*

***Initialisation (16)*** : Plus de 50% des *CVE* que nous avions analysées font partie des problèmes d’initialisation. Ce qui est intéressant ici c’est que TOUS les langages de programmation dans notre analyse ont eu au moins 1 *CVE* dans cette catégorie. Le tiers des *CVE* sont en écrit en *C*, moins du tiers des *CVE* écrit en *C* font partie de cette catégorie. La majorité des *CVE* font partie de la sous-catégorie *Predictable sequence (12)*. Ces résultats ont du sens car les problèmes de réutilisation de valeurs et de valeurs faibles sont plus faciles à prévenir, ceux-ci sont généralement causés par un manque de connaissance sur le processus de cryptographie. La même chose peut être dite par rapport aux erreurs de séquences prédictibles, par contre il y a plus d’éléments qui entrent en jeu. Le taux d’erreur devient beaucoup plus élevé car ces types de vulnérabilités ne sont pas nécessairement causés par des fonctions purement cryptographiques, ils sont causés par une mauvaise utilisation des fonctions aidantes (*helper functions*) utilisées dans le processus de cryptographie. Une connaissance générale du processus de cryptographie est aussi nécessaire pour s’assurer de ne pas commettre une erreur qui entre dans cette catégorie. Connaître les différentes composantes (*seed*, *salt*, *sufficient entropy*, etc.) qui doivent être utilisées dans le processus cryptographique est une chose, comprendre comment bien les utiliser est beaucoup plus difficile.

***Insecure Defaults (2)*** : Ce type de vulnérabilité est aussi facile à reproduire que réparer. Ce type d’erreur dépend largement de la documentation accessible au programmeur. Pour classer une vulnérabilité dans cette catégorie, l’utilisation d’une classe ou d’une méthode doit avoir une valeur par défaut qui est considéré comme étant pas assez cryptographiquement sécure. Elle se manifeste spécifiquement lorsque le polymorphisme paramétrique est impliqué, en autres mots, une fonction va avoir des comportements différents dépendamment du nombre et/ou du type d’arguments utilisées Si une classe ou une fonction utilise des valeurs par défauts insécure lorsqu’une séquence spécifique d’arguments est utilisées, ceci peut être problématique. Par contre, si la documentation est claire sur les faiblesses de la valeur par défaut, elle est définitivement évitable. Alors deux aspects contribuent à l’apparition de cette vulnérabilité :

* Utilisateurs de librairie : Le temps dépensé à regarder la documentation avant de l’utiliser dans un milieu professionnel.
* Créateurs de la librairie : Assurer une documentation de haute qualité dans le programme pour assurer que les utilisateurs utilisent la librairie de façon sécuritaire dans l’environnement respectif.

***Other (4)*** : La complexité associées à cette catégorie varie beaucoup. Les créateurs du *framework Tafelsalz* ont créer cette catégorie pour englober les vulnérabilités comme « ne pas utiliser la cryptographie du tout, les canaux secondaires introduites par les optimisations du compilateur, un traitement incomplet des tampons internes ou la suppression des secrets de la mémoire » [1] (traduction libre). Parmi les *CVE* analysées, seulement la *CVE-2016-10530* est considérée comme simple à comprendre et résoudre. Les autres s’appliquent tous au traitement incomplet des tampons internes. Ce type d’erreur peut être extrêmement difficile à comprendre car une erreur ne se produit pas tout le temps, dans le cas du *CVE-2014-3570*, l’erreur se produit avec une probabilité de 1/ sur une plateforme d’architecture *MIPS* de 32 bits et une probabilité de 1/ sur une plateforme quelconque de 64 bits. Spécifiquement dans ce cas, la vulnérabilité peut juste se manifester dans un contexte académique, il n’y a pas de réel danger (avec la présente technologie) au niveau industriel. C’est important de mentionner que ces 3 cas sont écrits en langage *C*. En d'autres mots, 100% des cas qui ont un problème de traitement incomplet des tampons internes (et qui ne sont pas des problèmes d’initialisation) sont écrits en *C*.

***Usage complexity (5)*** : Les vulnérabilités faisant partie de catégories sont certainement les plus difficiles à comprendre. Pour certaines techniques de cryptographie, leur utilisation est seulement sécure suite à une séquence spécifique d’opérations ayant des paramètres précis. Pour donner une idée de quoi peut ressembler une séquence spécifique d’opération, nous allons utiliser le *CVE-2017-7526*. La vulnérabilité se produit spécifiquement avec l’utilisation de la cryptographie asymétrique *RSA*. Lorsque la *méthode de gauche à droite* est utilisée pour calculer *l’expansion de la fenêtre glissante* [31], il y a une fuite d’information qui est exploitable par un acteur malicieux. Pour rendre cette technique sécure, le blindage d’exposant doit être utilisé pour ne brouiller le nombre de multiplications/exposants utiliser dans le calcul qui comprend la clé privé. D’ailleurs, même avec le blindage d’exposant, les implémentations qui ne sont pas calculées en temps constant ont tout de même des fuites d’informations. Même si un développeur maîtrise bien les concepts et les techniques de cryptographie qu’il utilise dans son programme, une simple erreur d’inattention peut rendre le code vulnérable. Les *CVE* analysées appartenant à cette catégories nous ont permis de comprendre davantage quelles sont les failles qu’un acteur malicieux pourrait rechercher pour agrandir son vecteur d’attaques.

***Validation (3)*** : Cette catégorie réfère au problème qui se situe dans la couche transport d’un protocole cryptographique. Il y a plusieurs façons qu’une vulnérabilité peut se manifester dans cette catégorie, par exemple l’utilisation de clés expirées, le manque de cryptage identifié, les problèmes en lien avec la validation d’un certificat *TLS*, etc. Les problèmes peuvent être obscurs car les vulnérabilités se produisent généralement à cause d’un traitement manquant ou non considéré dans les procédures impliqués.

Les catégories qui n’ont pas eu de *CVE* dans notre recherche sont :

1. *Initialization*
   1. *Source*
2. *Weak Algorithms*
3. *Persistence of Secrets*
4. *Knowledge Base*
5. *Identity Management*
6. *Password Hashing*

# **Outils d’analyse**

## ***Bandit* : *Python***

*Bandit* [34] est l’outil que nous avons utilisé pour analyser le code écrit en *Python*. *Bandit* observe les fonctions utilisées ainsi que les librairies importées et indiquera à l’utilisateur à quelle ligne ou lignes de code l’erreur est détectée et comment la remplacer.

* ***Initialisation (2)***
  + ***Predictable sequence (2)***
    - *CVE-2012-2417 : Python*
    - *CVE-2013-1445 : Python*
* ***Usage complexity (1)***
  + *CVE-2020-26263 : Python*
* ***Weak Algorithms (1)***
  + *CVE-2012-3458 : Python*

## ***Visual Code Grepper : C, C++, Java, PHP***

*Visual Code Grepper* [33] est l’outil que nous avons utilisé pour analyser le code en *C*, *C++*, *Java* et *PHP*. Le code observe les fonctions utilisées comme *GOTO* et donne un avertissement qu’il s’agit d’une pratique malsaine. S’il trouve des commentaires avec *TODO*, il donnera un avertissement que le code risque de ne pas être encore complet. Lorsque des lignes de code contiennent des mots comme “*password*” suivi de strings, des avertissement de mots de passes codés à la main sont donnés. Lorsque des ressources sont créées et ne sont pas effacées par la suite, il se peut que des attaques de type *Denial Of Service* (*DoS*) soient utilisées à l’aide de consommation excessive de ressources. De plus, lorsque des fonctions comme *random*, *java.util.random* ou *math.random* sont utilisées un avertissement sera émis afin d’avertir l'utilisateur que ces fonctions ne sont pas fiables pour créer des nombres aléatoires. Cela mène à la génération de nombres pseudo-aléatoires qui peuvent être déduits et mettre des utilisateurs à risque. Plusieurs fonctions utilisées font aussi partie de la liste de fonctions que *Microsoft* recommande de ne pas utiliser puisqu’elles sont faibles et à risque [31].

* ***Initialisation (10)***
  + ***Predictable sequence (6)***
    - *CVE-2011-0766 : Erlang, C*
    - *CVE-2014-5386 : C++*
    - *CVE-2015-8867 : C*
    - *CVE-2018-12520 : C++*
    - *CVE-2019-11808 : Java*
    - *CVE-2020-12735 : PHP*
* ***Re-use (2)***
  + *CVE-2019-15075 : PHP*
  + *CVE-2022-1434 : C*
* ***Weak Values (2)***
  + *CVE-2019-10908 : Java*
  + *CVE-2022-1235 : PHP*
* ***Insecure Defaults (1)***
  + *CVE-2016-1000352&1000344 : Java*
* ***Other (3)***
  + *CVE-2013-2548 : C*
  + *CVE-2014-3570 : C*
  + *CVE-2014-8275 : C*
* ***Usage complexity (2)***
  + *CVE-2017-7526 : C*
  + *CVE-2018-16870 : C*
* ***Validation (2)***
  + *CVE-2016-2053 : C*
  + *CVE-2019-11578 : C*

## ***DeepScan : JavaScript, TypeScript***

*DeepScan* [35] est l’outil qui a été utilisé pour identifier les vulnérabilités dans le code écrit en *JavaScript* et en *TypeScript*. Il est un programme gratuit à utiliser pour les projets open source, et il coûte de l’argent pour les projets privés. La façon dont il fonctionne est différente des autres outils, il analyse le code directement à partir des projets sur *GitHub*. En d'autres mots, les vulnérabilités ont été classifiées et téléchargées dans *GitHub* pour être en mesure d’analyser les vulnérabilités en *JavaScript* et *TypeScript*.

* ***Initialisation (2)***
  + ***Predictable sequence (2)***
    - *CVE-2021-41117 : JavaScript*
    - *CVE-2022-36045 : JavaScript*
* ***Other (1)***
  + *CVE-2016-10530 : JavaScript*
* ***Usage complexity (1)***
  + *CVE-2019-9155 : JavaScript*
* ***Validation (1)***
  + *CVE-2021-32738 : TypeScript (JavaScript)*

## **Staticcheck : Go**

*Staticcheck* [36] est l’outil qui a été utilisé pour analyser le code écrit dans le langage *Go*. De la même façon que les autres outils, il ressort les résultats d’analyse avec la ligne à laquelle une faute est détectée ainsi que le type d’erreur qui lui est attribué.

***Initialisation (2)***

* ***Predictable sequence (2)***
  + *CVE-2020-28924 : Go*
  + *CVE-2021-3538 : Go*
* ***Usage complexity (1)***
  + *CVE-2018-19653 : Go*

# **Résultats des outils d’analyses**

Une fois que nous avons passé les *CVE* dans les outils, les résultats obtenus étaient très intéressants.

## **Analyse du code vulnérable**

1. *CVE 2011-0766* : faux négatif, plusieurs avertissements de *buffer overflow* ont été émis
2. *CVE 2012-2417* : faux négatif, plusieurs avertissements de librairies qui ne sont plus supportées et sont à risque
3. *CVE-2012-3458* : faux négatif, avertissement de la librairie *pycrypto* et *AES* qui ne sont plus supportés
4. *CVE-2013-1445* : faux négatif
5. *CVE-2013-2548* : faux négatif, avertissement *d’integer overflow* et de *buffer overflow*
6. *CVE-2014-3570* : faux négatif, avertissement de *buffer overflow*
7. *CVE 2014-5386* : faux négatif, avertissement de *buffer overflow*
8. *CVE 2014-8275* : faux négatif
9. *CVE 2015 8867* : faux négatifs, avertissement d’integer overflow et de buffer overflow (avertissement critique)
10. *CVE 2016 2053* : faux négatif
11. *CVE-2016-10530* : faux négatif
12. *CVE 2016 1000344&1000352* : faux négatif
13. *CVE-2017-7526* : faux négatif
14. *CVE 2018 12520* : false negative, *avertissement d’integer* *overflow*
15. *CVE 2018 16870* : faux négatif
16. *CVE-2018-19653* : faux négatif
17. *CVE-2019-9155* : faux négatif
18. *CVE 2019 10908* : faux négatif, avertissement de mauvaise validation d’entrée
19. *CVE 2019 11578* : faux négatif, avertissement de *buffer overflow* dû au fait que des entrées externes sont directement assignées à un *buffer*
20. *CVE 2019 11808* : faux négatif
21. *CVE 2019 15075* : faux négatif
22. *CVE 2020 12735* : avertissement d’algorithme non-sécuritaire de hachage (techniquement il n’identifie pas la vulnérabilité)
23. *CVE 2020 26263* : faux négatif
24. *CVE-2020-28924* : faux négatif
25. *CVE-2021-3538* : faux négatif
26. *CVE-2021-32738* : faux négatif
27. *CVE-2021-41117* : faux négatif
28. *CVE-2022-36045* : faux négatif
29. *CVE 2022 1235* : avertissement d’algorithme non-sécuritaire, avertissement de PRNG (valid)
30. *CVE 2022 1434* : faux négatif

## **Analyse du code réparer**

1. *CVE-2011-0766* : vrai négatif, avertissement de *buffer overflow*
2. *CVE-2012-2417* : vrai négatif, plusieurs avertissements de librairies qui ne sont plus supportées et sont à risque
3. *CVE-2012-3458* : vrai négatif, avertissement de la librairie *pycrypto* et *AES* qui ne sont plus supportés
4. *CVE-2013-1445* : vrai négatif, avertissement de la librairies qui ne sont plus supportés
5. *CVE-2013-2548* : vrai négatif, avertissement *d’integer overflow* et de *buffer overflow*
6. *CVE-2014-3570* : vrai négatif, avertissement de *buffer overflow*
7. *CVE-2014-5386* : vrai négatif, avertissement de *buffer overflow*
8. *CVE-2014-8275* : vrai négatif
9. *CVE-2015-8867* : vrai négatif, avertissement *d’integer overflow* et de *buffer overflow* (avertissement critique)
10. *CVE-2016-2053* : vrai négatif
11. *CVE-2016-10530* : vrai négatif
12. *CVE-2016-1000352* : vrai négatif
13. *CVE-2017-7526* : vrai négatif
14. *CVE-2018-12520* : vrai négatif, avertissement *d’integer overflow*
15. *CVE-2018-16870* : vrai négatif
16. *CVE-2018-19653* : vrai négatif
17. *CVE-2019-9155* : vrai négatif
18. *CVE-2019-10908* : vrai négatif, avertissement de mauvaise validation d’entrée
19. *CVE-2019-11578* : vrai négatif, avertissement de *buffer overflow* dû au fait que des entrées externes sont directement assignées à un *buffer*
20. *CVE-2019-11808* : vrai négatif
21. *CVE-2019-15075* : vrai négatif
22. *CVE-2020-12735* : vrai négatif, aucun avertissement d’algorithme de hachage
23. *CVE-2020-26263* : vrai négatif
24. *CVE-2020-28924* : vrai négatif
25. *CVE-2021-3538* : vrai négatif
26. *CVE-2021-32738* : vrai négatif
27. *CVE-2021-41117* : vrai négatif
28. *CVE-2022-1235* : vrai négatif
29. *CVE-2022-1434* : vrai négatif
30. *CVE-2022-36045* : vrai négatif

# **Analyse des résultats**

## **Informations primaires**

### **Versions défaillantes**

De nos 30 *CVE*, seulement la *CVE-2022-1235* a été détectée par les outils. Il fait partie de la catégorie de vulnérabilité *Initialisation - Weak Values*. Dans le cas de *CVE 2022-1235*, il s’agit d’un hachage assez faible pour qu’il soit deviné à partir d’attaque de force brute (environ 16^10 possibilités). Il s’agit donc d’un problème de valeurs faibles. Lors de l’analyse de la *CVE* avec l’outil *Visual Code Grepper*, un avertissement est émis mettant l’utilisateur en garde que l’utilisation de certaines fonctions sont non sécuritaires puisque ces fonctions génèrent des nombres pseudo-aléatoires qui sont déterministes et prévisibles. Cela permet alors à un attaquant d’énumérer les valeurs possibles jusqu’à ce que la bonne soit trouvée.

Graphical user interface, text, application, email

Description automatically generated

*Figure 4. Résultats d’analyse du code vulnérable du CVE-2022-1235*

### **Versions corrigées**

Pour la *CVE-2022-1235*, elle a par la suite donné un vrai négatif lors de l’analyse de la version corrigée du code. Une fois que le code corrigé est passé dans *Visual Code Grepper*, aucune erreur n’est soulevée par l’outil puisque le *fix* introduit produit 16^80 possibilités de *secrethash* au lieu de 16^10 ce qui est beaucoup plus sécuritaire. L’outil est alors capable de détecter le *fix* effectué dans le code *lhc+web\_modules\_lhinstall\_install.php*.

## **Informations secondaires**

### **Versions Défaillantes**

Parmi les 30 *CVE* que nous avons utilisés, plusieurs d’entre elles, même si elles ont donné un faux négatif, ont tout de même émis des avertissements mettant les utilisateurs en garde de code erroné. Il vaut la peine de mentionner que presque chaque *CVE* écrites en *C/C++* contenait des *GOTO*. Même si ce n’est pas un problème en soit, les *GOTO* peuvent mal structurer le code et rendre l’allocation et désallocation de mémoire difficile. Nous avons eu plusieurs avertissements de niveau moyen mettant en garde que certaines fonctions comme *memcpy* en *C* peuvent faciliter des *buffer overflow* dans certaines situations ainsi que des *integer overflow*. La fonction *memcpy* fait partie de la liste de fonctions mises à l’index par *Microsoft* alors c’est un avertissement qu’il faudrait prendre en considération. Pour d'autres *CVE*, nous avons aussi eu des avertissements mettant des utilisateurs en garde contre l’utilisation de certaines librairies comme la librairie *pyCrypto* qui n’est plus maintenue et peut devenir un risque à long terme.

Plus sérieusement, nous avons eu des avertissements pour des *CVE* comme *CVE-2020-12735*, *CVE-2019-11578* et *CVE-2015-8867* ont reçu des avertissements sévères et critiques mettant en garde les utilisateurs des algorithmes de hachage faibles et de débordement de tampon. Dans le cas de *CVE-2019-11578*, l’utilisation de la fonction *fscanf* est ce qui a alerté l’outil. Cette fonction fait partie des fonctions mises à l’index par *Microsoft*, puisqu’elle dirige des entrées externes dans un tampon et ouvre la porte à des attaques de débordement de tampon.

Text

Description automatically generated

*Figure 5. Résultats d’analyse de la version défaillante de CVE-2019-11578*

Quant au *CVE-2015-8867*, elle fait appel à la fonction *strlcpy* qui copie un caractère et l’assigne à un tampon de taille fixe. Des personnes malintentionnées peuvent alors utiliser cette fonction à leur avantage en créant un *buffer overflow* pour ensuite s’en prendre au reste du système.

Graphical user interface, text, application, email

Description automatically generated

*Figure 6. Résultats d’analyse de la version défaillante de CVE-2015-8867*

Pour ce qui est de *CVE-2020-12735*, il s’agit de l’algorithme de hachage *MD5* qui est considéré comme faible et peut être brisé en quelques secondes. Il est alors déconseillé d’utiliser ces algorithmes dans un contexte de sécurité informatique.

Graphical user interface, text, application, email

Description automatically generated

*Figure 7. Résultats d’analyse de la version défaillante de CVE-2020-12735*

### **Versions corrigées**

Les résultats du code des versions corrigées ont beaucoup varié. Premièrement, puisque les *GOTO* ne sont pas des erreurs cryptographiques ou portent un risque sérieux au code, ils se retrouvent encore dans le code et des avertissements ressortent. Cependant, des avertissements de fonctions, comme *memcpy* ou *strncpy*, indexées par *Microsoft* se retrouvent encore dans les analyses des versions corrigées. Bien que ces fonctions soient bannies et non-recommandées par le titan de l’industrie, si ces fonctions ne sont utilisées que dans des classes ou bouts de code non critiques, il n’y a pas grand risque de *buffer overflow*. De plus, des avertissements de librairies expirées se retrouvent encore dans l’analyse de certaines *CVE*. C’est entièrement normal. Ces *CVE* ont été répertoriées au cours de plusieurs années et certaines librairies ont évolué ou ont entièrement été recréées afin de rester utiles et sécuritaires. Alors, lorsque la correction de la *CVE* est sortie, il se peut que l’utilisation de ces librairies fût encore entièrement sécuritaire et même recommandée.

Cependant, des fonctions permettant l'interférence directe avec des buffers ne devraient pas être implémentées tout simplement. En prenant par exemple *CVE-2019-1578* et *CVE-2015-8867*, les mêmes avertissements ont été émis. Considérant qu’il s’agit d’avertissements sévères et critiques dû à des fonctions qui sont fortement déconseillées pour des raisons de manipulation directe entre un utilisateur externe et le système, observer l’impact que ces fonctions et classes ont dans le code serait recommandé afin de déterminer si ces risques de buffer overflow sont sérieux.

Bien que l’utilisation de l’algorithme de hachage MD5 n’était pas la source de la faille impliquée dans la CVE-2020-12735, il a tout de même été enlevé lors de la correction du code à la prochaine version. Cela veut dire qu’il aurait pu être source de problèmes dans le futur et les développeurs ont décidé d’implémenter une version alternative de hacher des valeurs afin de prévenir des failles dans le futur.

# **Conclusion**

Les vulnérabilités cryptographiques sont parmi celles qui sont les plus dévastatrices dans le monde réel. Le but de ce rapport était de cataloguer et catégoriser 30 *CVE* qui était en lien avec la cryptographie en utilisant une taxonomie pré déterminer [1]. Ensuite, nous voulions évaluer les résultats d’outils d’analyse cryptographique utilisant une méthode d’analyse statique pour déterminer s’ils étaient capables d’identifier ces vulnérabilités.

Parmi les *CVE* que nous avons analysés, 8 différents langages de programmation ont été utilisés. Les données acquises de l’échantillonnage que nous avons effectué nous permettront d’avoir une idée sur les types d’erreurs cryptographiques les plus communes dans la communauté, les résultats sont très pertinents. Plus de 50% des *CVE* se retrouvent dans la catégorie de type de vulnérabilité *Initialisation*. La majorité se retrouvaient dans la sous-catégorie *Predictable Sequences*. Chaque langage de programmation avait au moins 1 *CVE* dans cette catégorie. Ceci veut dire que ce type de vulnérabilité se manifeste facilement d’un langage à l’autre. Ceci n’est pas vrai pour les autres catégories, par exemple la catégorie *Usage Complexity* est la deuxième catégorie contenant le plus de *CVE*, seulement la moitié (4) des langages sont parmi celle-ci.

Les outils d’analyses ont repéré seulement 1 *CVE*, elle a été catégorisée comme étant une vulnérabilité de type *Initialisation – Weak Values*. On a été surpris de ne pas avoir plus de vrais positifs parmi les résultats d’analyse des version de code vulnérable car plusieurs des *CVE* consistait de l’utilisation d’une fonction problématique. La méthode d’analyse statique est efficace contre les signatures de fonctions, pourtant les résultats obtenus ne reflètent pas cette réalité. C’est certain que l’analyse statique ne permettrait pas de trouver efficacement des problèmes situées dans des catégories plus complexes (*Other*, *Usage Complexity*), mais elle devrait être efficace contre les vulnérabilités dans la catégorie *Initialisation*.

Il serait intéressant d’évaluer la performance des outils d’analyse cryptographique utilisant les méthodes dynamique et interactive. Il serait aussi intéressant de voir comment on peut intégrer des changements à un outil d’analyse pour qu’il soit en mesure de trouvées plus de vulnérabilités appartenant à une catégorie spécifique. En combinant les résultats des deux recherches, il serait possible de significativement diminuer le potentiel d’erreurs dans des protocoles cryptographiques.

Finalement, le processus d’acquisition des données *CVE* est reconnu comme étant un travail manuel et laborieux. Après avoir développé une méthode qui a été relativement efficace pour trouver le code source en lien avec des *CVE* dans une catégorie *CWE* spécifique, on est confiant qu’un outil pourrait être développé pour diminuer significativement la charge de travail manuel requise dans ce processus. Les listes *CVE* et *CWE* apporte une certaine ambigüité, mais malgré leur ambiguïté, ils donnent l’opportunité d’avoir accès à des données utilisées dans le marché à travers le temps. La pertinence d’avoir une telle application dans le domaine de recherche serait très bénéfique pour la communauté, avec plus de 200 000 *CVE* existant, les possibilités de recherche sont très vastes.

# Bibliographie

[1] Blochberger, M., Petersen, T., & Federrath, H. (2019). Mitigating cryptographic mistakes by design. *Mensch und Computer 2019-Workshopband*.

[2] Red Hat Bugzilla Main Page. (s. d.). Consulté le 16 octobre 2022, à l’adresse <https://bugzilla.redhat.com/>

[3] Home - Vulncode-DB. (s. d.). Consulté le 17 septembre 2022, à l’adresse <https://www.vulncode-db.com/>

[4] Snyk Vulnerability Database | Snyk. (s. d.). Find detailed information and remediation guidance for vulnerabilities. Consulté le 16 octobre 2022, à l’adresse <https://security.snyk.io/>

[5] CWE - Common Weakness Enumeration. (s. d.). Consulté le 16 octobre 2022, à l’adresse <https://cwe.mitre.org/index.html>

[6] GitHub - secureIT-project/CVEfixes : CVEfixes : Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. (s. d.). GitHub. Consulté le 19 septembre 2022, à l’adresse <https://github.com/secureIT-project/CVEfixes>

[7] cve-website. (s. d.). Consulté le 16 octobre 2022, à l’adresse <https://www.cve.org/About/Overview>

[8] Honkaranta, A., Leppänen, T., & Costin, A. (2021, May). Towards practical cybersecurity mapping of stride and cwe—a multi-perspective approach. In 2021 29th Conference of Open Innovations Association (FRUCT) (pp. 150-159). IEEE.

[9] Tenable Community. (s. d.). Consulté le 16 octobre 2022, à l’adresse <https://community.tenable.com/s/question/0D5f2000053XWzgCAG/multiple-cve-values-for-the-same-vulnerability>

[10] Sridhar, K., Householder, A., Spring, J., & Woods, D. W. (2021, June). Cybersecurity Information Sharing: Analysing an Email Corpus of Coordinated Vulnerability Disclosure. In The 20th Annual Workshop on the Economics of Information Security.

[11] NVD - CWE Layout. (s. d.). Consulté le 17 octobre 2022, à l’adresse <https://nvd.nist.gov/vuln/categories/cwe-layout>

[12] A02 Cryptographic Failures - OWASP Top 10 : 2021. (s. d.). Consulté le 17 octobre 2022, à l’adresse <https://owasp.org/Top10/A02_2021-Cryptographic_Failures/>

[13] CVE security vulnerability database. Security vulnerabilities, exploits, references and more. (s. d.). Consulté le 16 octobre 2022, à l’adresse <https://www.cvedetails.com/>

[14] NVD - Home. (s. d.). Consulté le 16 octobre 2022, à l’adresse <https://nvd.nist.gov/>

[15] Heffley, J., & Meunier, P. (2004, January). Can source code auditing software identify common vulnerabilities and be used to evaluate software security?. In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the* (pp. 10-pp). IEEE.

[16] Eclipse Foundation. (s. d.). CogniCrypt - Secure Integration of Cryptographic Software | CogniCrypt. Securely using Cryptography with CogniCrypt. Consulté le 18 octobre 2022, à l’adresse https://www.eclipse.org/cognicrypt/

[17] Cryptosense. (s. d.). Consulté le 18 octobre 2022, à l’adresse <https://cryptosense.com/>

[18] GitHub - spotbugs/spotbugs : SpotBugs is FindBugs’ successor. A tool for static analysis to look for bugs in Java code. (s. d.). GitHub. Consulté le 18 octobre 2022, à l’adresse <https://github.com/spotbugs/spotbugs>

[19] Egele, M., Brumley, D., Fratantonio, Y., & Kruegel, C. (2013, November). An empirical study of cryptographic misuse in android applications. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 73-84).

[20] Piccolboni, L., Di Guglielmo, G., Carloni, L. P., & Sethumadhavan, S. (2021, May). Crylogger: Detecting crypto misuses dynamically. In 2021 IEEE Symposium on Security and Privacy (SP) (pp. 1972-1989). IEEE.

[21] Blessing, J., Specter, M. A., & Weitzner, D. J. (2021). You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries. *arXiv preprint arXiv:2107.04940*.

[22] Lazar, D., Chen, H., Wang, X., & Zeldovich, N. (2014, June). Why does cryptographic software fail? A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (pp. 1-7).

[23] What Is SAST and How Does Static Code Analysis Work ? | Synopsys. (s. d.). <https://www.synopsys.com/glossary/what-is-sast.html>

[24] What is Dynamic Application Security Testing (DAST) | Micro Focus. (s. d.). <https://www.microfocus.com/en-us/what-is/dast>

[25] Veracode. (s. d.). What is IAST ? Interactive Application Security Testing. <https://www.veracode.com/security/interactive-application-security-testing-iast>

[26] Oracle Critical Patch Update Advisory - October 2020. (s. d.). <https://www.oracle.com/security-alerts/cpuoct2020.html>

[27] Time to patch : Vulnerabilities exploited in under five minutes ? (2022, 8 avril). Infosec Resources. <https://resources.infosecinstitute.com/topic/time-to-patch-vulnerabilities-exploited-in-under-five-minutes/>

[28] CWE - PDFs with Graphical Depictions of CWE (Version 4.9). (s. d.). <https://cwe.mitre.org/data/pdfs.html>

[29] GitHub : Let’s build from here. (s. d.). GitHub. <https://github.com/>

[30] Dashboard. (s. d.). DeepScan. <https://deepscan.io/dashboard/>

[31] Howard, M. H. [x509cert]. (2022, 19 avril). Banned/banned.h. GitHub. Consulté le 18 décembre 2022, à l’adresse <https://github.com/x509cert/banned/blob/master/banned.h>

[32] Bernstein, D. J., Breitner, J., Genkin, D., Groot Bruinderink, L., Heninger, N., Lange, T., ... & Yarom, Y. (2017, September). Sliding right into disaster: Left-to-right sliding windows leak. In International Conference on Cryptographic Hardware and Embedded Systems (pp. 555-576). Springer, Cham.

[33] nccgroup. (s. d.-b). GitHub - nccgroup/VCG : VisualCodeGrepper - Code security scanning tool. GitHub. <https://github.com/nccgroup/VCG>

[34] Welcome to Bandit — Bandit documentation. (s. d.). <https://bandit.readthedocs.io/en/latest/>

[35] How to ensure JavaScript code quality. (s. d.). DeepScan. <https://deepscan.io/>

[36] Honnef, D. (s. d.). GitHub - dominikh/staticcheck-action : Staticcheck’s official GitHub Action. GitHub. <https://github.com/dominikh/staticcheck-action>