Annexe 1 :
Descriptif des failles et des outils

Mourad Djellouli
https://github.com/Mourad7/SolidityCode

Projet de session INF889A - Analyse de programmes pour la sécurité logicielle –

Jean Privat

Table des matières

Fonctionnement des différents outils (Approches, stratégies) 2
A. SmartCheck : 2
B. Slither : 2
C. Manticore : 2
D. Securify : 3
E. Myhtx : 3
F. Autres outils : 4
Echidna : 4
Solidity Visual Auditor : 4
Solhint : 4
G. Récapitulatif : 4
Collecte des instances 5
Description des failles 6
1. Visibilité des fonctions et variables par défaut (SWC100/SWC 108) Type S 6
2. Retrait d'éther non protégé (SWC 105) Type S 6
3. Autorisation via tx.origin (SWC 115) Type S : 6
4. Utilisation des fonctions Solidity dépréciées (SWC 111) Type S : 6
5. Version de compilateur flottante type S : 7
6. Réentrance type S : 7
7. Dépendance de l'horodatage (Timestamp SWC 116) Type BC : 7
8. Faibles sources de génération d'attributs aléatoires (SWC 120) Type BC : 7
9. Attaque d'adresse courte type EVM : 8
10. Ether perdu type EVM : 8
11. Rubixi Type PS : 8
Tableau récapitulatif des vulnérabilités choisies : 8

Fonctionnement des différents outils (Approches, stratégies)

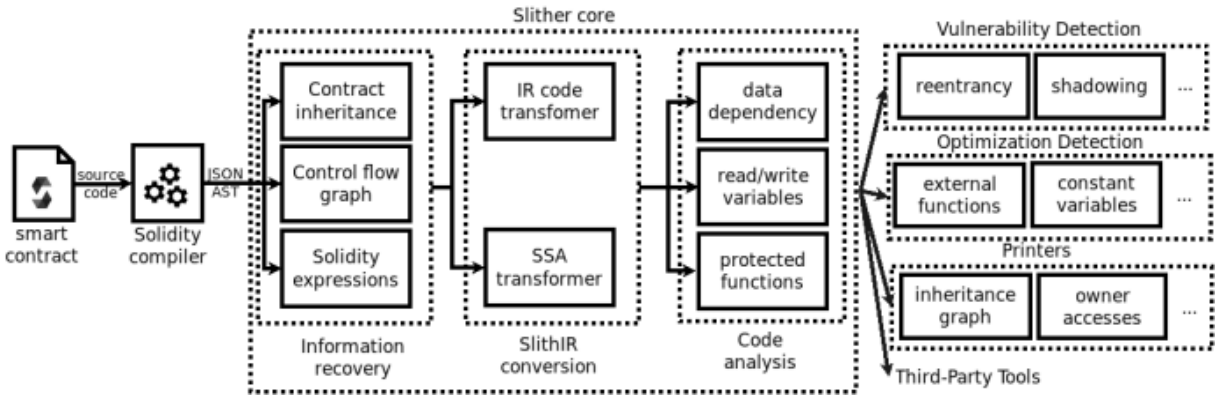
A. SmartCheck :

Analyseur statique de code pour les contrats écrits en Solidty. Développé par SmartDec et l'université du Luxembourg. SmartCheck signale les vulnérabilités potentielles dans les contrats écrits dans le langage Solidity, en recherchant des patterns syntaxiques dans le code source. Ensuite, procède en convertissant le code dans un arbre syntaxique XML. Les vulnérabilités sont spécifiées en tant qu'expressions de chemin XQuery utilisées pour rechercher les modèles dans l'arborescence XML. L'outil est écrit en Java et est disponible en deux versions :

Une version en ligne de commande est disponible sur Github sous licence GPL-3.0 depuis mai 2017, accompagnant le document académique original. La version la plus récente de l'outil avec environ deux fois plus de nombreux modèles sont des sources fermées et sont accessibles via le site Web de la société SmartCheck, qui a également contient une liste des problèmes de sécurité.

B. Slither :

Analyseur statique pour le langage Solidity écrit en Pyhton 3. Développé par Trail of Bits, nécessitant toutefois que les contrats soient écrits au moins en Solidity 0.4 (>=). Slither analyse les contrats en utilisant l'analyse statique procèdent en plusieurs étapes. Il prend comme entrée initiale l'Arbre de syntaxe abstraite (AST) généré par le compilateur Solidity à partir du code source du contrat. Dans un premier temps, Slither récupère des informations importantes telles que le graphe d'héritage du contrat, le graphique de flux de contrôle (CFG) et la liste des expressions. Ensuite, Slither transforme tout le code du contrat à SlithIR, son langage de représentation interne. SlithIR utilise static single assessment (SSA) pour faciliter le calcul d'une variété d'analyses de code. Au cours de la troisième étape, Slither calcule un ensemble d'analyses prédéfinies qui vont fournir des informations améliorées aux autres modules.



C. Manticore :

Manticore utilise une exécution symbolique pour trouver des chemins de calcul uniques dans les binaires EVM (et ELF). À l'aide du solveur SMT Z3, il trouve des entrées qui déclencheront ces chemins de calculs. Il enregistre les traces d'exécution correspondantes. Concernant l'EVM, Manticore compile le code Solidity en bytecode pour son analyse, vérifie les traces de vulnérabilités comme la réentrance et les opérations d'autodestruction et les signale dans le contexte du code source. Informations sur les méthodes et leur les limitations sont rares.

L'outil est développé et maintenu par la société Trail of Bits, et disponible sur GitHub sous licence AGPL-3.0 depuis février 2017. Il peut être utilisé à partir d'une CLI ou via une API Python.

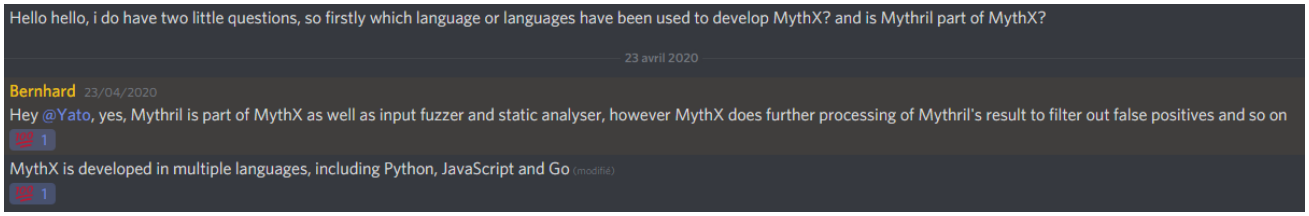
D. Securify :

Analyseur créé par des chercheurs de l'ETH Zurich en collaboration avec ChainSecurity, fonctionne à la fois sur les fichiers sources de Solidity, mais également les Bytecodes. Il commence par analyser le contrat de manière symbolique pour obtenir des informations sémantiques. Puis vérifie les modèles pour voir si une propriété de sécurité tient ou non. Une interface Web est également disponible.

Securify prend en entrée le bytecode EVM et les propriétés de sécurité. L'outil décompile la pile orientée bytecode dans un formulaire basé sur une affectation et représente le code en tant que faits DataLog. Ensuite, il dérive d'autres faits qui décrivent flux de contrôle de données dans un résumé forme. Une propriété de sécurité consiste en la conformité et schéma de violation surestimant les deux, satisfaction et non-satisfaction de ce bien. Les motifs sont codés en tant que règles DataLog qui peuvent être vérifiées par rapport au fait en utilisant Soufflé. Cette approche garantit que si un le modèle est détecté, le code possède / viole définitivement la propriété de sécurité correspondante. L'outil est écrit en Java et disponible sur GitHub sous une licence Apache2.0 depuis septembre 2018. De plus, une version fermée la version source est accessible sur le site Web de la société ChainSecurity. Rien n'indique la différence entre les deux versions.

E. Myhtx :

Outil non-open source, développé par ConsenSys, disponible via des extensions (VS Code, Truffle, Remix, CLI) les résultats de l'analyse sont consultables via un Dashboard en ligne. Il a fallu contacter un développeur de chez ConsenSys pour avoir plus d'informations :



MythX contient plusieurs analyseurs dont un statique et un de fuzzing et Mythril qui est open-source, qui utilise une combinaison d'exécution symbolique et de Taint analysis pour identifier les failles de sécurité.

Contrairement aux autres outils il est écrit en plusieurs langages dont Python, JavaScript et Go,

F. Autres outils :

Echidna :

Libraire Haskell également développé par Trail of Bits, permettant de faire du Fuzzing (injection de données aléatoires), reposant sur le principe du teste basé sur les propriétés du code EVM, Echidna a pour objectif de générer des inputs pouvant faire crasher les contrats intelligents.

Echidna faisait partie des outils choisis dans le cadre de la comparaison, seulement après l'avoir essayé plusieurs fois, il a été incapable de trouver des inputs pouvant mener à des chemins d'exécution différents. Sans doute dû au fait que les failles étudiées ne dépendent pas des inputs.

Solidity Visual Auditor :

Extension pour l'éditeur Visual Studio Code, outil de visualisation, qui a pour objectif de nous aider à mieux comprendre le code Solidity en offrant plusieurs vues :

- Les diagrammes UML,
- Les graphes de Surya et des rapports,
- Les call graphs (pour voir les dépendances entre les fonctions).
- Les graphes d'héritage
- Une représentation en texte de l'AST

Solhint :

Linter open source pour le langage Solidity, ils en existent d'autres, mais c'est le seul qui est encore maintenu à ce jour, il est disponible sur plusieurs IDE tel que Visual Studio Code.

G. Récapitulatif :

	SmartCheck	**Slither**	**Securify**	**Manticore**	**MyhtX**
Principes de bases	Patterns Arbre XML XQuery	AST CFG Héritage	Datalog Facts CFG	Solveur SMT Z3	Non-open source
Langage	Java	Pyhton 3	Java	Pyhton 3	?
Version	2.0.1	0.6.11	1.0	0.3.3	0.6.12
Type d'analyse	Analyse statique	Analyse statique	Exécution symbolique	Exécution symbolique	Analyse statique Exécution symbolique
WUI/CLI	WUI + CLI	CLI	WUI + CLI	CLI	WUI + CLI
Src-file / ByteCode	Src-file	Src-file	Src-file + Bytecode	Src-file + Bytecode	Src-file + Bytecode
Développé par	SmartDec	Trail of Bits	ETH Zurich	Trail of Bits	ConsenSys

Collecte des instances

Une des premières difficultés rencontrées pour cette étape, est le fait d'installer les différents outils, sachant que certains sont disponibles sur Windows d'autres non, peuvent dépendre de npm ou Pypi. Et pour pouvoir comparer les outils « équitablement » il est important de les faire tourner sur les mêmes conditions (configuration) partout, j'ai opté pour la solution de conteneurs Docker, qui n'est pas malheureusement disponible pour tous les outils et à cause de cette contrainte supplémentaire la comparaison du temps d'exécution n'a plus réellement de sens.

- SmartCheck / Securify : Version Web (contenant plus d'analyseurs)
- Slither / Manticore / Echidna : conteneur Docker
- MythX : Extension disponible sur VS Code

Après avoir choisi les outils et les présenter, il a fallu identifier les différentes failles à étudier, dont les descriptions sont présentes sur le GitHub du projet, mais également récolter des contrats intelligents contenant des instances de ces failles, et cela depuis plusieurs sources :

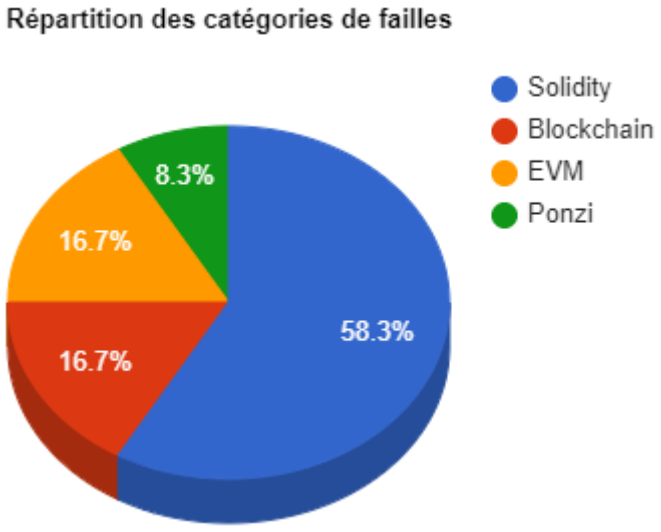
- Smart Contract Weakness Classification (SWC): https://swcregistry.io/
- Not-so Smart Contracts: https://github.com/crytic/not-so-smart-contracts

Ensuite, j'ai procédé à la vérification du code de chacun des contrats pour mieux comprendre ce qu'ils faisaient, il fallait également définir une version de compilateur fixe pour tous les contrats dans le cadre du Benchmark.

Systèmes de Ponzi : Il a été observé dans le passé que certains contrats intelligents d'Ethereum ont étaient utilisés pour créer des systèmes de Ponzi, pour arnaquer des utilisateurs innocents en leur promettant des gains énormes sur leurs investissements. Même si en soi un système Ponzi ne représente pas une vulnérabilité de sécurité directe, j'ai jugé pertinent de les rajouter à cause de la menace que cela peut représenter pour l'argent des utilisateurs, mais également sur la crédibilité de la technologie Blockchain en général.

On se retrouve ainsi avec 4 catégories principales de failles réparties comme suit :

A. **Type S :** Failles dues à une mauvaise utilisation du le langage Solidity.
B. **Type EVM :** Failles engendrées durant la compilation par a machine EVM.
C. **Type BC :** Failles présentes dans l'architecture de la Blockchain Ethereum en général
D. **Type PS :** Systèmes de Ponzi (Ponzi Schemes)



Description des failles

1. Visibilité des fonctions et variables par défaut (SWC100/SWC 108) Type S

Les fonctions et les variables qui n'ont pas de type de visibilité sont publiques par défaut. Cela peut causer une vulnérabilité si le développeur oublie de la spécifier et un utilisateur malintentionné décide d'apporter des changements d'état non autorisés ou involontaires.

Solution : Spécifier la visibilité comme étant Private, Internal, External, Public :

1) **Private** : une fonction / variable privée ne peut être appelée que dans le contrat dans laquelle elle a été déclarée (contrat source).
2) **Internal** : la fonction/ variable peut être appelée depuis le contrat source, mais également depuis les contrats dérivés
3) **External** : la fonction / variable peut être appelée uniquement depuis un contrat extérieur qui aucune relation avec notre contrat source, ni ses dérivés
4) **Public** : **le mode par défaut** des fonctions / variables, qui spécifie que cette dernière peut être appelée depuis toutes les parties potentielles.

Output : l'analyseur devrait être capable de nous dire quelles fonctions / variables n'ont pas de visibilité par défaut.

2. Retrait d'éther non protégé (SWC 105) Type S

En raison de contrôles d'accès manquants ou insuffisants, les parties malveillantes peuvent retirer tout ou partie de l'Ether du compte du contrat.

Ce bogue est parfois provoqué par l'exposition involontaire de fonctions d'initialisation. En nommant à tort une fonction destinée à être un constructeur, le code du constructeur se retrouve dans le code d'octet d'exécution et peut être appelé par n'importe qui pour réinitialiser le contrat.

Solution : Mettre en œuvre des contrôles afin que les retraits ne puissent être déclenchés que par des parties autorisées ou selon les spécifications du système de contrat intelligent.

3. Autorisation via tx.origin (SWC 115) Type S :

Tx.origin est une variable globale dans Solidity qui renvoie l'adresse du compte qui a envoyé la transaction. Utiliser cette variable pour des autorisations pourrait rendre un contrat vulnérable si jamais un compte autorisé appelle un contrat malveillant.

Solution : Ne pas utiliser tx.origin pour des autorisations, mais plutôt msg.sender

4. Utilisation des fonctions Solidity dépréciées (SWC 111) Type S :

Utiliser des fonctions dépréciées peut mener à réduire la qualité du code. Et cela peut même engendrer des comportements inattendus avec les nouvelles versions des compilateurs Solidity.

Solution : Suivre la documentation à jour, et remplacer les fonctions dépréciées (par exemple on peut remplacer sha3 avec keccak256).

5. Version de compilateur flottante type S :

Les contrats doivent être déployés avec la même version du compilateur et les mêmes indicateurs avec lesquels ils ont été testés de manière approfondie. Le verrouillage du Pragma permet de garantir que les contrats ne sont pas déployés accidentellement en utilisant, par exemple, une version de compilateur obsolète qui pourrait introduire des bogues affectant négativement le système de contrats.

Solution : Verrouiller la version Pragma et tenir compte des bogues connus pour la version du compilateur choisie

6. Réentrance type S :

L'un des principaux dangers d'appeler des contrats externes est qu'ils peuvent prendre en charge le flux de contrôle. Dans l'attaque de réentrance (aka attaque d'appel récursive), un contrat malveillant rappelle dans le contrat appelant avant la fin du premier appel de la fonction. Cela peut provoquer des interactions indésirables entre les différentes invocations de la fonction.

Solution : Les meilleures pratiques pour éviter les faiblesses de réentrance sont les suivantes :

1. Assurez-vous que tous les changements d'état internes sont effectués avant l'exécution de l'appel. Ceci est connu comme le modèle de contrôles-effets-interactions
2. Utilisez un verrou de réentrance (c.-à-d. ReentrancyGuard d'OpenZeppelin)

7. Dépendance de l'horodatage (Timestamp SWC 116) Type BC :

Les contrats ont souvent besoin d'accéder à l'horodatage actuel pour déclencher certains événements. À cause de la nature décentralisée d'Ethereum, les nœuds ne peuvent synchroniser l'heure que dans une certaine mesure. Ainsi les mineurs malveillants peuvent en profiter pour modifier l'horodatage de leur bloc. (Soit pour enclencher des événements plus tôt que prévu, ou au contraire les empêcher).

Solution : Les développeurs ne devraient pas faire confiance à la précision de l'horodatage fourni, et plutôt envisager l'utilisation des numéros des blocs ou encore des sources externes d'horodatage via des oracles.

8. Faibles sources de génération d'attributs aléatoires (SWC 120) Type BC :

Utilisé dans plusieurs cas d'applications, on génère souvent des nombres aléatoires, cependant créer une source de nombres aléatoires suffisamment forte dans Ethereum est très difficile, par exemple, l'utilisation de block.timestamp n'est pas sécurisée, car un mineur peut choisir de fournir n'importe quel horodatage en quelques secondes et de faire accepter son bloc par d'autres. Si les enjeux sont élevés, le mineur peut créer beaucoup de blocs en peu de temps en louant du matériel, choisir le bloc nécessaire, et laisser tomber tous les autres.

Solution :

- Utiliser d'un schéma d'engagement, par exemple RANDAO
- Utiliser des sources externes pour la génération aléatoire (via des oracles), par exemple Oraclize.
- Utiliser des hachages de blocs Bitcoin, car ils sont plus robustes

9. Attaque d'adresse courte type EVM :

Faille découverte par Golden Team, permettant à l'attaquant d'abuser la fonction de transfert. L'EVM rajoute des zéros si la longueur de l'adresse est inférieure à la longueur nécessaire

Solution : vérifier (via un assert par exemple) si la taille du message est supérieure ou égale.

10. Ether perdu type EVM :

Quand on veut envoyer de l'Ether (Monnaie de Ethereum), on doit spécifier l'adresse du destinataire (160 bits). Seulement plusieurs de ces adresses sont dites « orphelines », c'est-à-dire non-associé à un utilisateur ou contrat. Et si on en envoie à une de ces adresses, l'Ether est perdu à tout jamais.

Solution : il n'y pas de moyen de détecter si une adresse est orpheline ou non, les développeurs devront s'en assurer manuellement.

11. Rubixi Type PS :

Rubixi est un contrat qui met en œuvre un schéma de Ponzi :

Au cours de l'élaboration du contrat, son nom *Dynamic Pyramid a* été changé en *Rubi*xi. Cependant, le nom du constructeur n'a pas été modifié en conséquence par les développeurs. Cette fonction est alors devenue invocable par n'importe qui, le hack permet à un adversaire de voler de l'éther du contrat. La fonction Dynamic Pyramid définit l'adresse du propriétaire ; le propriétaire peut retirer son bénéfice via **collectables.** Après que ce bug soit devenu public, les utilisateurs ont commencé à invoquer Dynamic Pyramid afin de devenir le propriétaire, et ainsi de retirer les fonds.

```
1  contract Rubixi {
2    address private owner;
3    function DynamicPyramid() { owner = msg.sender; }
4    function collectAllFees() { owner.send(collectedFees); }
5    ...
```

Tableau récapitulatif des vulnérabilités choisies :

Permettant de regrouper les vulnérabilités qu'on va traiter et les classifier selon leurs types, ainsi que leurs impacts sur la sécurité :

Vulnérabilités	Classification (Type)	Impact sur la sécurité
Visibilité par défaut (SWC 100-108)	Type S	Elevé
Autorisation via tx.origin (SWC 115)	Type S	Critique
Retrait d'éther non protégé (SWC 105)	Type S	Critique
Utilisation de fonctions dépréciées (SWC 111)	Type S	Faible
Version du compilateur flottante (SWC 103)	Type S	Faible
Réentrance (SWC 107)	Type S	Élève
Dépendance de l'horodatage (SWC 116)	Type BC	Élève
Faible génération aléatoire (SWC 120)	Type BC	Critique
Attaque d'adresse courte	Type EVM	Elevé
Ether perdu	Type EVM	Moyen