Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«17» травня 2024 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)
спеціальність *125 Кібербезпека*
(код і назва спеціальності)
освітній ступень *магістр*
освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Захист від вразливостей SWC-100-136»

Виконавець: студент ІІ курсу, групи КБм-21

_____ **Ярослав КУЛАГА**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Сергій ТОЛЮПА	
Нормоконтроль	Яніна ШЕСТАК	

Київ 2024

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«17» листопада 2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача(ки) КБм-21 (група) Кулага Ярослав Сергійович (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Захист від вразливостей SWC-100-136

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 5 від 15.11.2023 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Процес захисту від вразливостей SWC-100-136.
Предмет досліджень	Методи та методики оцінки захисту від вразливостей смарт-контрактних.
Мета	Ретельний аналізу безпеки смарт-контрактів та створення сканера вразливостей смарт-контрактів
Вихідні дані для проведення роботи	Методи захисту від вразливостей SWC-100-136 у смарт-контрактах.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна — Вирішення науково-прикладної задачі підвищення захищеності смарт-контрактів, а саме написання методики для покращення безпеки смарт-контракту та написання сканеру для виявлення вразливостей в смарт-контрактах.

Практична цінність — Розробка рекомендацій захисту від вразливостей SWC-100-136 для безпезпеки смарт-контрактів та розробка сканеру смарт-контрактів для пошуку вразливостей SWC-100-136.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Робота виконана у повному обсязі відповідно до теми.

5. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	17.09.2023 – 24.09.2023
Аналіз літературних джерел вразливостей SWC-100, 102, 103, 104	25.09.2023 – 29.09.2023
Написання коду для сканування вразливостей SWC-100, 102, 103, 104	30.09.2023 – 08.10.2023
Аналіз літературних джерел	10.10.2023 – 01.11.2023
Ознайомлення з сучасними смарт-контрактів	02.11.2023 – 21.11.2023
Розгляд нормативно-правових актів, регулюючих функціонування та захист інформації щодо смарт-контрактів	22.11.2023 – 06.12.2023
Розробка рекомендацій для безпеки смарт-контрактів	07.12.2023 – 01.02.2024
Дослідження вразливостей SWC-100-136	02.02.2024 – 25.02.2024
Написання коду для сканування вразливостей SWC-100-112	25.02.2024 – 15.03.2024
Відладка коду для сканування вразливостей SWC-100-112	16.03.2024 – 19.03.2024
Написання коду для сканування вразливостей SWC-113-124	20.03.2024 – 01.04.2024
Відладка коду для сканування вразливостей SWC-113-124	01.04.2024 – 05.04.2024

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Написання коду для сканування вразливостей SWC-125-136	01.06.2024 – 15.04.2024
Відладка коду для сканування вразливостей SWC-125-136	16.06.2024 – 18.04.2024
Відладка коду для сканування вразливостей SWC-100-136	18.04.2024 – 21.04.2024
Оформлення пояснювальної записки згідно методичних рекомендацій	22.04.2024 – 12.05.2024
Подача пакету документів на розгляд ЕК	13.05.2024 – 17.05.2024

6. РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект Зниження збитків через покращення безпеки смарт-контракту

Соціальний ефект Покращення захисту від вразливостей SWC-100-136 як особисто так і на підприємствах.

7. ДОДАТКОВІ ВИМОГИ

Завдання видав _____ Сергій ТОЛЮПА
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Ярослав КУЛАГА
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 17.09.2023 р.
Термін подання кваліфікаційної роботи до ЕК 17.05.2024 р.

РЕФЕРАТ

Пояснювальна записка містить 93 сторінки та 67 літературних джерел.

Об'єкт дослідження - сканер вразливостей, розроблений спеціально для смарт-контрактів блокчейну.

Мета роботи - ретельний аналізу безпеки смарт-контрактів та створення сканера вразливостей смарт-контрактів, яким зможуть користуватися розробники та аудитори, а також корисних порад.

Для того, щоб знайти можливі слабкі місця в смарт-контрактах і посилити їхню безпеку, в рамках дослідження були створені і вивчені методи сканування. Для вибору найкращого інструменту для створення сканера було проведено порівняння мов програмування, зокрема Solidity, Vyper та Yul.

Наступним етапом стало вивчення теоретичних недоліків, що лежать в основі смарт-контрактів, знайдених в SWC-100 - SWC-136. У світлі цього дослідження було створено унікальний сканер смарт-контрактів, який пройшов тестування в реальному світі, щоб знайти ці недоліки і підвищити безпеку смарт-контрактів.

Результати дослідження продемонстрували, що створений сканер є більш успішним, ніж існуючі на сьогоднішній день методи виявлення недоліків та підвищення безпеки смарт-контрактів.

Ключові слова: СМАРТ-КОНТРАКТИ, SWC-100-136, СКАНЕР СМАРТ-КОНТРАКТІВ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

SWC	–	Smart Contract Weakness Classification
PSA	–	Payment Services Act
CAES	–	Crypto Asset Exchange Services
EVM	–	Ethereum virtual machine
CEI	–	Check-echo-interactio
DEX	–	Decentralized exchange

ЗМІСТ

ВСТУП

Сучасність свідчить про стрімкий розвиток смарт-контрактів, які змінюють наше уявлення про цифрові угоди. Поглиблене дослідження їхнього потенціалу відкриває нам широкі перспективи та виклики. Ethereum, як відома блокчейн-платформа, відіграє ключову роль у цьому процесі, каталізуючи розвиток смарт-контрактів та їхнє впровадження в різні галузі. Наша робота спрямована на дослідження смарт-контрактів Ethereum та забезпечення їхньої безпеки, щоб сприяти розвитку цифрової екосистеми, де безпека та довіра є першочерговими.

Об'єкт, предмет, мета та завдання дослідження

Об'єкт дослідження - процес захисту від вразливостей SWC-100-136.

Предмет дослідження - методи та методики оцінки захисту від вразливостей смарт-контрактних.

Мета роботи - ретельний аналізу безпеки смарт-контрактів та створення сканера вразливостей смарт-контрактів, яким зможуть користуватися розробники та аудитори, а також корисних порад.

Завдання:

- Зробити пошук та аналіз рекомендацій, стандартів, наукової літератури.
- Зробити пошук та аналіз законодавчих баз різних країн щодо захисту смарт-контракту.

Наукова новизна та практичне значення результатів

Наукова новизна роботи полягає у вирішенні науково-прикладної задачі підвищення захищеності смарт-контрактів, а саме написання методики для покращення безпеки смарт-контракту та написання сканеру для виявлення вразливостей в смарт-контрактах.

РОЗДІЛ 1
НОРМАТИВНО-ПРАВОВА БАЗА, ІСТОРІЯ СТВОРЕННЯ ТА МАЙБУТНЄ СМАРТ-КОНТРАКТІВ ETHEREUM

Для розробників, аудиторів, пентестерів та користувачів дуже важливо мати чітке розуміння нормативно-правового середовища, що стосується смарт-контрактів. Ця інформація необхідна для гарантування безпечного та надійного впровадження смарт-контрактів. Дуже важливо вивчити минулий розвиток і сучасні тенденції смарт-контрактів, щоб спрогнозувати їхній майбутній курс і прогрес.

1.1. Аналіз та пошук стандартів безпеки смарт-контрактів

У пошуках універсального стандарту безпеки смарт-контрактів стало очевидним, що єдиного загальноприйнятого стандарту не існує. Однак галузь розробила безліч передових практик, цінних ресурсів і рекомендацій експертів, які пропонують розробникам міцну основу для створення надійних і безпечних смарт-контрактів.

У світлі відсутності єдиного стандарту, наше розуміння важливості використання цих передових практик і цінних ресурсів як наріжного каменю для розробки нашого стандарту безпеки смарт-контрактів ще більше зросло. Такий підхід дозволяє нам максимально використовувати колективну мудрість експертів галузі та випробувані стратегії для забезпечення найвищого рівня безпеки і цілісності під час розробки смарт-контрактів.

Розуміння, що безпека смарт-контрактів є ключовим фактором для зміцнення довіри до екосистеми блокчейн, виробляє у нас рішучий погляд на завдання. Ми прагнемо не просто створити стандарт, а створити всеосяжний, надійний і адаптований стандарт безпеки, який стане маяком для розробників, підприємств і зацікавлених сторін. Наш стандарт буде сприяти впровадженню найкращих у своєму класі практик безпеки, давати розробникам можливість створювати стійкі смарт-

контракти і, в кінці кінців, сприятиме створенню більш безпечного та надійного середовища блокчейну.

1.2 Аналіз та пошук законів щодо безпеки смарт-контрактів в Україні та світі

Правове поле України щодо смарт-контрактів на стадії становлення. Станом на сьогодні, 17 березня 2024 року, не існує чіткого законодавства, яке б безпосередньо регулювало аспекти безпеки смарт-контрактів.

У минулому спостерігалися спроби регулювати криптовалюти та технологію блокчейн. Зокрема, у 2017 році 6 жовтня до парламенту було внесено законопроект "Про обіг криптовалют в Україні" (№ 7183) [11]. Цей законопроект мав на меті забезпечити нормативно-правову базу для використання криптовалют в Україні. Однак він не був прийнятий.

Незважаючи на відсутність цільового регулювання безпеки смарт-контрактів, постійний розвиток правової бази України свідчить про потенціал для майбутніх змін у цій сфері, оскільки використання технології блокчейн та смарт-контрактів продовжує розширюватися.

Наразі на міжнародному рівні не існує універсальних законів, спеціально спрямованих на безпеку смарт-контрактів.

Законодавча база Японії щодо безпеки смарт-контрактів наразі є неповною. Станом на 17 березня 2024 року не існує жодного конкретного закону, який би безпосередньо стосувався цього питання.

Однак є деякі відповідні нормативні акти, які стосуються смарт-контрактів:

Закон про платіжні послуги (PSA): Цей закон регулює послуги з обміну криптовалют (Crypto Asset Exchange Services, CAES), які працюють з токенами, випущеними за допомогою технології блокчейн, з аналізу [12].

Незважаючи на відсутність всеосяжної правової бази, Японія продовжує брати участь в дискусіях та ініціативах, спрямованих на розширення використання

технології блокчейн і смарт-контрактів, залишаючи відкритою можливість для подальшого розвитку в цій сфері.

Сполучене Королівство зайняло унікальну позицію щодо безпеки смарт-контрактів. На відміну від багатьох країн, станом на 17 березня 2024 року у Великій Британії немає спеціальних законів, що безпосередньо стосуються безпеки смарт-контрактів. Замість цього Великобританія застосувала інноваційний підхід, адаптувавши існуючу правову базу до смарт-контрактів.

У 2021 році Комісія з питань права Великої Британії дійшла висновку, що чинне договірне право Англії та Уельсу може бути застосоване до смарт-контрактів [Law Commission, Smart contracts] [13]. Це означає, що для оцінки смарт-контрактів використовуються встановлені правові принципи щодо формування та виконання контрактів.

Такий підхід має кілька переваг:

- **Юридична гнучкість:** Використання існуючої правової бази забезпечує гнучкість у вирішенні проблем, пов'язаних зі смарт-контрактами, що швидко розвиваються.
- **Підвищена передбачуваність:** Застосування встановлених правових принципів до смарт-контрактів надає бізнесу більший ступінь визначеності, сприяючи створенню більш впевненого і стабільного бізнес-середовища.

Стратегія Великобританії відображає далекоглядне прагнення привести правовий ландшафт у відповідність до технологічного прогресу, зберігаючи при цьому видимість стабільності і ясності для підприємств, які беруть участь у розгортанні смарт-контрактів.

Станом на 17 березня 2024 року Австралія, як і багато інших країн, ще не прийняла спеціальне законодавство, безпосередньо спрямоване на безпеку смарт-контрактів. Однак існують правові рамки та галузеві дискусії, які впливають на цю сферу:

Відповідні закони:

Закон про електронні транзакції (1999): Цей закон визнає дійсність і можливість примусового виконання електронних контрактів.

Наше комплексне дослідження показало, що станом на 17 березня 2024 року жодна країна, включно з Україною, не має спеціального законодавства, що регулює безпеку смарт-контрактів. Реакція регуляторних органів у різних країнах світу різниться, причому деякі юрисдикції, такі як Велика Британія та Австралія, вирішили адаптувати існуючі правові рамки, щоб охопити характерні ознаки та наслідки смарт-контрактів.

У світлі зростаючої популярності та значення смарт-контрактів на світовій економічній арені, для бізнесу та юристів-практиків вкрай важливо залишатися пильними та відслідковувати законодавчі зміни у цій сфері. Така уважність дозволить їм випереджати нові регуляторні тенденції, ефективно управляти потенційними юридичними ризиками та використовувати можливості, що виникають завдяки постійному розвитку технології смарт-контрактів.

1.3. Коротко про смарт-контракт Ethereum

Що робить смарт-контракт "розумним"? Технологія, що лежить в основі смарт-контрактів, - це блокчейн. Простіше кажучи, блокчейн - це база даних, яка розподілена між декількома комп'ютерами. Коли відбуваються транзакції, записи про них об'єднуються в блоки коду, які додаються до ланцюжка (реєстру транзакцій). Головною перевагою цієї структури, яка є загальнодоступною і використовує складну криптографію для забезпечення безпеки та достовірності записів, є те, що вона усуває потребу в централізованому органі, який би якимось чином сприяв або виступав посередником при проведенні транзакцій. Жоден банк не бере участі у встановленні тарифів або переказі коштів; безпека і розподіл системи замінює цей вид зовнішнього посередницького контролю. [1]

"Смарт-контракт" - Смарт-контракти - це програми, які керують поведінкою облікових записів в межах мережі Ethereum.[2]

Смарт-контракти - це цифрові контракти, що самостійно виконуються, в яких умови угоди між сторонами виражені безпосередньо в рядках програмного коду. Іншими словами, смарт-контракт - це програмне забезпечення, яке автоматично

запускається і забезпечує дотримання правил і умов угоди між двома або більше сторонами. У сфері блокчейну він має форму комп'ютерної програми, що зберігається на вузлі мережі блокчейн і може бути запрограмована на автоматичний запуск дій, транзакцій або інших подій на основі попередньо встановлених умов. [3]

Порівнявши ці визначення ми можемо визначення ми можемо зрозуміти що таке смарт-контракт та зробити своє визначення яке поєднує найкраще з вище перелічених варіантів та не змінює сенсу.

Смарт-контракт - це частина програмного коду, розміщена на блокчейні Ethereum. Він включає в себе функції та дані, що зберігаються за унікальною адресою в мережі блокчейн. Ці цифрові контракти працюють автономно і виконують заздалегідь визначені дії, як тільки виконуються певні умови, закодовані в них.

Вони зазвичай використовуються для автоматизації виконання угоди, щоб усі учасники могли одразу бути впевнені в результаті, без участі посередників і без втрати часу. Вони також можуть автоматизувати робочий процес, запускаючи наступну дію, коли виконуються заздалегідь визначені умови.

Ethereum - це розподілений блокчейн, який може виконувати смарт-контракти, що взаємодіють між собою і виконують транзакції автоматично. Виконання смарт-контрактів оплачується у вигляді газу, який є грошовою одиницею, що використовується в блокчейні Ethereum. Віртуальна машина Ethereum (EVM) забезпечує можливість обліку виконання смарт-контрактів. Вартість інструкцій варіюється в залежності від типу інструкції та приблизних обчислювальних ресурсів, необхідних для виконання інструкції в мережі. Вартість газу коригується за допомогою комісій за транзакції, щоб забезпечити адекватну оплату мережі[4].

Наприклад, уявімо, що Аліса хоче купити квартиру від Боба. Вони укладають угоду, але для безпеки обидва бажають мати впевненість, що угода виконається коректно.

Замість того, щоб використовувати традиційні методи, такі як переказ коштів через банк або використання адвокатів, Аліса та Боб можуть скористатися смарт-контрактом на блокчейні.

Угода в смарт-контракті може включати умови, такі як:

Передача власності: Квартира автоматично переходить на Алісу, як тільки вона внесе певну суму коштів.

Відкладений платіж: Смарт-контракт може утримувати кошти до тих пір, поки не будуть виконані умови, такі як підписання договору про купівлю-продаж, здача-приймання квартири тощо.

Відмова від угоди: Якщо угода скасовується з будь-яких причин, кошти можуть автоматично повертатися Алісі.

Смарт-контракт забезпечить обох сторін безпекою і впевненістю, що угода буде виконана відповідно до умов, без потреби довіряти третій стороні, такій як банк або адвокат. Це також може зменшити витрати і скоротити час на укладення угоди.

Застосування смарт-контрактів в повсякденному житті [5]:

- автоматизація типових бізнес-процесів
- фінансові операції
- демократичне децентралізоване управління
- управління ланцюгами поставок
- реєстрація нерухомості, авторських прав

Отже, смарт-контракти Ethereum відіграють важливу роль у формуванні майбутнього децентралізованих систем, революціонізуючи взаємодію і бізнес-процеси завдяки надійним, безпечним і автоматизованим рішенням.

1.4 Історія створення смарт-контрактів Ethereum

Зародження смарт-контрактів Ethereum можна простежити з перших днів розвитку технології блокчейн і навіть раніше, як частину ширшого бачення створення децентралізованого та автоматизованого світу. Ця історична подорож починається з появи комп'ютеризованих протоколів для автоматизації контрактів у 1990-х роках, що призвело до революційної розробки Ethereum у 2015 році, яка перетворила концепцію смарт-контрактів на практичну реальність.

У цей період ранній технологічний прогрес уможливив появу елементарних форм смарт-контрактів. Одним з яскравих прикладів є DigiCash, платіжна система,

орієнтована на конфіденційність, розроблена Чаумом та ін. у 1988 році. Ця система використовувала криптографію та цифрові підписи для створення цифрової валюти, яку можна було анонімно передавати між користувачами, що заклало основу для майбутніх рішень для цифрових контрактів.

Починаючи з 1990-х років, комп'ютерні вчені та математики створювали технічні інструменти для автоматизації контрактів. Більш того, деякі технології, що дозволяли створювати ранні та рудиментарні форми смарт-контрактів, вже існували в той час, коли Сабо ділився своїми думками зі світом. Прикладом такої технології є DigitCash Чаума та ін. (1988), платіжна система, яка захищала конфіденційність користувачів[6].

У 1994 році Нік Сабо представив смарт-контракти, які є комп'ютерними програмами, що відтворюють дії, описані у фізичних/традиційних контрактах. Пізніше, в 1996 році, Сабо визначив наступні цілі смарт-контракту: спостережуваність, верифікованість, пріоритетність і можливість примусового виконання [7].

Пізніше, у 2015 році, Віталік Бутерін створив Ethereum, блокчейн-платформу, яка має децентралізовану платіжну систему і повну мову Тьюринга, що дозволяє розробляти широкий спектр смарт-контрактів на блокчейні[8].

Ethereum був офіційно запущений у 2015 році, зробивши смарт-контракти доступними для всього світу. Відтоді він став провідною платформою для створення децентралізованих додатків з використанням смарт-контрактів. Ці самодостатні контракти, що самостійно виконуються та застосовуються, суттєво вплинули на такі галузі, як фінанси, охорона здоров'я, нерухомість та управління ланцюгами поставок, пропонуючи підвищену прозорість, ефективність та довіру.

Успіх Ethereum продовжує стимулювати інновації в блокчейн-просторі, що призвело до розробки Ethereum 2.0, також відомої як Serenity. Це оновлення має на меті покращити масштабованість, безпеку та стійкість мережі Ethereum

Цей крок виявився критичним у розвитку технології, оскільки Ethereum став платформою, де смарт-контракти набули широкого застосування, від фінансових послуг до галузі supply chain та управління ланцюгами поставок.

На завершення, історія смарт-контрактів Ethereum позначена візіонерами, які передбачили їхній потенціал, і технологічними досягненнями, які зробили їх реальністю. Від свого зародження в ранніх протоколах автоматизації контрактів до широкого впровадження на платформі Ethereum, смарт-контракти зробили революцію в індустрії і продовжують формувати майбутнє децентралізованих систем.

1.5 Майбутнє смарт-контрактів Ethereum

Майбутнє смарт-контрактів Ethereum виглядає багатообіцяючим, з потенціалом до широкого впровадження в різних галузях. Ось деякі з ключових факторів, які сприяють цьому:

- **Зростаюча масштабованість:** Ethereum вирішує проблеми масштабованості завдяки впровадженню рішень другого рівня та переходу на Ethereum 2.0. Це дозволить платформі обробляти більше транзакцій швидше та дешевше, роблячи смарт-контракти більш практичними для широкого використання. Ethereum, провідна світова блокчейн-платформа для децентралізованих додатків, готується до загальносистемного оновлення, яке докорінно змінить спосіб використання та захисту платформи. Це довгоочікуване оновлення, що отримало назву Ethereum 2.0, розробники ядра Ethereum працюють над ним з моменту першого релізу платформи в 2015 році. За останні п'ять років відбулося кілька невеликих оновлень, які покращили зручність використання та масштабованість Ethereum. Однак Ethereum 2.0 - це, безумовно, найбільш амбітна і радикальна зміна, яка буде впроваджена в мережі, і на її повну реалізацію знадобиться кілька років [9].
- **Підвищена зручність використання:** Розробляються інструменти та платформи, які роблять створення та використання смарт-контрактів більш доступними для людей з меншим технічним досвідом. Це може призвести до буму інновацій та нових застосувань смарт-контрактів.

- **Широке прийняття:** Все більше компаній та організацій досліджують та впроваджують смарт-контракти на платформі Ethereum. Це свідчить про зростаючу довіру та визнання цінності цієї технології [10].

Деякі потенційні сфери застосування смарт-контрактів Ethereum включають:

- **Децентралізовані фінанси :** Смарт-контракти використовуються для створення децентралізованих аналогів традиційних фінансових послуг, таких як кредитування, позики та торгівля.
- **Управління ланцюжками постачання:** Смарт-контракти можуть автоматизувати та оптимізувати ланцюжки постачання, підвищуючи прозорість та ефективність.
- **Голосування:** Смарт-контракти можуть використовуватися для створення безпечних та прозорих систем голосування.
- **Управління ідентичністю:** Смарт-контракти можуть використовуватися для безпечного зберігання та управління даними про ідентичність.
- **Ігри:** Смарт-контракти можуть використовуватися для створення нових типів ігор та віртуальних активів.

Загалом, майбутнє смарт-контрактів Ethereum виглядає багатообіцяючим, з потенціалом до революційних змін у багатьох галузях. Зростаюча масштабованість, зручність використання та прийняття стимулюють інновації та створюють нові можливості для використання цієї потужної технології.

Висновки за розділом 1

Пошук універсального стандарту безпеки смарт-контрактів не привів до єдиного, широко прийнятного рішення. Однак галузь накопичила безліч передових практик, експертних думок і цінних ресурсів, які допомагають розробникам створювати надійні та безпечні смарт-контракти. Реакція світових регуляторних органів на безпеку смарт-контрактів залишається різноманітною: такі країни, як Великобританія та Австралія, адаптують свою законодавчу базу, щоб охопити цю технологію, що розвивається. Враховуючи зростаюче значення смарт-контрактів у

світовій економіці, бізнес та юристи-практики повинні пильно стежити за розвитком законодавства, щоб залишатися в авангарді регуляторних змін, успішно управляти потенційними юридичними ризиками та використовувати можливості, що випливають з безперервного розвитку технології смарт-контрактів.

У цьому розділі ми ретельно проаналізували низку визначень смарт-контрактів, підкресливши їхню адаптивність та потенціал. Зіставивши ці визначення, ми сформували наше розуміння смарт-контрактів як самодостатніх цифрових угод, що самостійно виконуються та забезпечують примусове виконання, які існують в межах децентралізованої мережі блокчейн.

Приклади з реального життя підкреслюють численні переваги смарт-контрактів, такі як підвищення ефективності, прозорості та довіри до транзакцій і процесів. Ці реальні сценарії є переконливим свідченням здатності смарт-контрактів оптимізувати операції та ефективно вирішувати перешкоди, з якими стикаються традиційні системи.

Крім того, ми здійснили глибоку ретроспективну подорож крізь багату історію смарт-контрактів Ethereum, простеживши їхній шлях від стадії зародження наприкінці 1980-х років до сучасних застосувань.

Отже, смарт-контракти Ethereum стали трансформаційною силою у сфері децентралізованих систем, очоливши зміну парадигми взаємодії та бізнес-процесів завдяки надійним, безпечним та автоматизованим рішенням, що не потребують довіри. Оскільки технологія блокчейн невпинно рухається вперед, смарт-контракти приречені залишатися в авангарді інновацій, стимулюючи трансформації в різних галузях і стимулюючи появу новаторських додатків. Дивлячись у майбутнє, стає очевидним, що подальший розвиток технології смарт-контрактів вимагатиме постійної адаптації та інновацій як від розробників, так і від регуляторів. Зберігаючи пильність, бізнес та юристи можуть гарантувати, що вони залишатимуться в змозі орієнтуватися в динамічному правовому середовищі та використовувати нові можливості. Зрештою, ці колективні зусилля з адаптації та інновацій сприятимуть зміцненню довіри та безпеки в екосистемі блокчейну, що розширюється.

РОЗДІЛ 2
РОЗРОБКА РЕКОМЕНДАЦІЙ ДЛЯ БЕЗПЕКИ СМАРТ-КОНТРАКТІВ

Уважно вивчивши переваги і варіанти використання кожної мови програмування, розробники можуть приймати обґрунтовані рішення про найбільш підходящу мову для свого сканера смарт-контрактів і смарт-контрактів. Цей вибір в кінцевому підсумку вплине на ефективність, продуктивність і безпеку кінцевого продукту.

2.1 Розуміння віртуальної машини Ethereum

Для ефективної розробки та розгортання безпечних смарт-контрактів на блокчейні Ethereum розробникам вкрай важливо отримати глибоке розуміння EVM, її архітектури та обмежень:

Віртуальна машина Ethereum є основою мережі Ethereum, служачи основою для виконання смарт-контрактів і децентралізованих програм. Він відіграє ключову роль у тому, щоб Ethereum став не просто платформою для криптовалют, розширюючи його можливості для підтримки широкого спектру децентралізованих сервісів і програм. Нижче ми детальніше розглянемо EVM, досліджуючи його архітектуру, функціональні можливості та значення у світі блокчейну та децентралізованих обчислень [14].

Архітектура EVM: EVM - це середовище виконання для виконання коду смарт-контрактів на блокчейні Ethereum. Воно працює на основі стекової архітектури і використовує певний набір інструкцій, відомих як опкоди, для виконання операцій. Розробники повинні розуміти різні компоненти EVM, такі як стек, пам'ять і сховище, а також те, як EVM обробляє виконання контрактів, виклики повідомлень і переходи станів.

Вартість газу: В Ethereum кожна операція, що виконується смарт-контрактом, витрачає певну кількість газу, який є мірою обчислювальних зусиль. Вартість газу

для різних опкодів можна знайти в Жовтій книзі Ethereum, і її слід враховувати під час розробки контракту, щоб мінімізувати використання газу і запобігти потенційним вразливостям, пов'язаним з газом.

Опкоди: Опкоди Ethereum представляють конкретні операції, які можуть бути виконані на EVM, такі як арифметика, зберігання даних і операції з управління потоками. Розуміння доступних опкодів, їх функціональності та вартості газу може допомогти розробникам оптимізувати свої смарт-контракти і уникнути потенційних проблем, пов'язаних з несподіваною поведінкою EVM.

Обмеження EVM і неочікувана поведінка: Деякі опкоди EVM можуть призвести до неочікуваної поведінки за певних умов. Наприклад, використання опкоду DELEGATECALL у резервній функції Solidity може призвести до атаки на вхід, якщо з ним не обережно поводитись. Розробники повинні знати про ці потенційні пастки і забезпечити стійкість свого коду до таких вразливостей.

Глибоко розуміючи віртуальну машину Ethereum, розробники можуть писати більш ефективні, безпечні та надійні смарт-контракти для Ethereum та інших EVM-сумісних блокчейнів.

2.2 Міркування щодо безпеки Solidity

Solidity, будучи найбільш широко використовуваною мовою для написання смарт-контрактів на Ethereum, має свій власний набір потенційних вразливостей безпеки. Щоб вирішити ці проблеми, розробники повинні слідувати встановленим кращим практикам і використовувати безпечні шаблони кодування.

Одна з критичних вразливостей, яка потребує ретельного розгляду, пов'язана з неперевіреними зовнішніми викликами, які становлять значну загрозу для цілісності контрактів Solidity. Виклики зовнішніх функцій, якщо ними не керувати належним чином, можуть слугувати потенційними точками входу для зловмисних експлойтів, підриваючи безпеку всієї системи.

Неперевірені зовнішні виклики: Виклики зовнішніх функцій можуть бути потенційним джерелом вразливостей в Solidity. Щоб зменшити ці ризики, розробники повинні:

- Уникати надсилання ефіру на невідомі адреси без належної перевірки.
- З обережністю використовувати низькорівневі функції, такі як call, delegatecall і staticcall, і завжди перевіряти значення, що повертаються, на успішність або неуспішність [15].

Шаблон повторного входу і взаємодія CEI представляють собою нюансовані, але критичні області вразливості в екосистемі смарт-контрактів Solidity, що вимагають всебічного розуміння і ретельних стратегій пом'якшення наслідків для забезпечення надійної безпеки. Атаки повторного входу, зокрема, становлять величезну загрозу, коли контракт ініціює виклик зовнішньої функції, яка рекурсивно повторно входить в контракт до завершення початкового виклику, що потенційно призводить до несанкціонованого маніпулювання станом контракту і використання критичних вразливостей.

Шаблон повторного входу та взаємодії "перевірка-ефект-вплив-взаємодія" (CEI): Атаки на повторний вхід можуть виникати, коли контракт викликає зовнішню функцію, яка потім повертається до початкового контракту до того, як початковий виклик завершиться. Щоб запобігти уразливостям типу "повторний вхід":

- Використовуйте шаблон Checks-Effects-Interactions (CEI), який передбачає відокремлення викликів зовнішніх функцій від оновлення станів у функціях контракту.
- Використовуйте модифікатор reentrancyGuard, що надається бібліотекою OpenZeppelin, щоб запобігти викликам, які можуть призвести до повторного входу.

Використання готових бібліотек: Щоб не вигадувати велосипед і не створювати нові вразливості, розробникам слід використовувати бібліотеки з хорошою репутацією в сфері безпеки, такі як:

- OpenZeppelin: Широко використовувана бібліотека для смарт-контрактів Solidity, яка надає безпечні, багаторазові компоненти і шаблони для різних випадків використання [16].
- SafeMath: Бібліотека, яка пропонує арифметичні операції з перевіркою на переповнення, запобігаючи потенційним вразливостям цілочисельного переповнення [17].
- Стандарти токенів ERC20 та ERC721: Усталені стандарти токенів, які визначають загальний набір правил та інтерфейсів для створення взаємозамінних (ERC20) та не взаємозамінних (ERC721) токенів [18][19].

По суті, безпека контрактів Solidity ґрунтується на ретельному впровадженні найкращих практик та розумному використанні безпечних шаблонів кодування. Неухильно дотримуючись цих принципів і зберігаючи пильність щодо потенційних вразливостей, розробники можуть зміцнити цілісність своїх контрактів, забезпечуючи надійність і стійкість екосистеми Ethereum.

Дотримуючись цих специфічних для Solidity практик безпеки, розробники можуть створювати більш безпечні смарт-контракти, знижуючи ризик експлойтів і непередбачуваної поведінки.

2.3 Взаємодія з сервісами Ethereum

При розробці смарт-контрактів та взаємодії з різними сервісами Ethereum, такими як гаманці та децентралізовані біржі, важливо дотримуватися безпечних практик, щоб захистити кошти користувачів та зберегти цілісність контракту.

Розробка смарт-контрактів передбачає не лише забезпечення їхньої функціональності, але й надання пріоритету безпеці коштів користувачів та цілісності самого контракту. При взаємодії з різними сервісами Ethereum, такими як гаманці та децентралізовані біржі, вкрай важливо дотримуватися ретельних практик безпеки.

Безпечна взаємодія з гаманцями:

- **Забезпечте авторизацію користувачів:** Завжди запитуйте у користувачів авторизацію при виконанні транзакцій або отриманні доступу до їхніх гаманців.

Використовуйте такі бібліотеки, як @metamask/detect-provider та @metamask/onboard, для виявлення та безпечного підключення до гаманців користувачів [20].

- **Заохочуйте найкращі практики:** Інформувати користувачів про важливість безпечного керування обліковими даними гаманців та використання апаратних гаманців для додаткової безпеки.
- **Уникайте непотрібних погоджень:** Обмежте вимогу контракту щодо дозволів на використання токенів до мінімальної суми, необхідної для його функціонування.

Захистіть взаємодію з DEX:

- **Мінімізуйте залежність від третіх сторін:** Зменшити залежність від сторонніх сервісів в рамках контракту, щоб мінімізувати потенційні вектори атак.
- **Використовуйте перевірені протоколи DEX:** Використовуйте добре зарекомендовані і ретельно перевірені протоколи DEX, такі як Uniswap або 0x, щоб забезпечити безпечну і надійну торгівлю [21].
- **Відстежуйте взаємодію між контрактами:** Регулярно переглядайте і контролюйте взаємодію між контрактом і DEX, щоб виявити і усунути будь-яку підозрілу активність або потенційні вразливості.

Мінімізація залежності від сторонніх послуг:

- **Перевірка даних у ланцюжку:** Коли це можливо, перевіряйте дані в межах самого контракту, а не покладайтеся на зовнішніх постачальників даних.
- **Децентралізовані оракули:** Якщо потрібні зовнішні дані, використовуйте децентралізовані мережі оракулів, такі як Chainlink, для безпечного отримання та перевірки даних [22].
- **Регулярний аудит та оновлення:** Регулярно перевіряйте та оновлюйте контракт, щоб вирішувати потенційні проблеми безпеки, які можуть виникнути через інтеграцію сторонніх сервісів.

Дотримуючись цих безпечних практик при взаємодії з сервісами Ethereum, розробники можуть створювати смарт-контракти, які є більш стійкими до атак і краще захищають активи та дані своїх користувачів.

2.4 Використання інструментів безпеки Ethereum

Щоб підвищити безпеку смарт-контрактів Solidity, розробники повинні використовувати доступні інструменти і методи для виявлення потенційних вразливостей і забезпечення коректності.

Інструменти статичного аналізу:

Mythril: Інструмент аналізу безпеки з відкритим вихідним кодом, який виявляє потенційні вразливості в контрактах Solidity, такі як цілочисельні переповнення, повторний вхід і необроблені виключення [23].

Mythril є незамінним інструментом аналізу безпеки з відкритим вихідним кодом, пристосованим для розробки смарт-контрактів Ethereum, який вміє виявляти потенційні вразливості, що ховаються в контрактах Solidity. Пропонуємо вам ознайомитися з можливостями та значенням Mythril:

- **Виявлення вразливостей:** Mythril спеціалізується на виявленні широкого спектру вразливостей, які становлять ризики для безпеки та цілісності контрактів Solidity. Ці вразливості охоплюють такі критичні проблеми, як цілочисельні переповнення, атаки на повторний ввід, необроблені винятки та багато інших. Ретельно аналізуючи код контракту та перевіряючи шляхи виконання, Mythril дозволяє розробникам виявляти приховані вразливості та захищати свої контракти від потенційних експлойтів.
- **Комплексний аналіз:** Mythril проводить ретельний і всебічний аналіз контрактів Solidity, використовуючи передові методи статичного аналізу і символьне виконання для вивчення всіх можливих шляхів виконання. Таке вичерпне дослідження дозволяє Mythril виявляти вразливості в різних сценаріях і граничних ситуаціях, забезпечуючи всебічне покриття і надійну оцінку безпеки. Незалежно від того, чи перевіряєте ви складні смарт-контракти

або аудит критично важливих кодових баз, Mythril надає безцінну інформацію для розробників, які прагнуть підвищити стійкість своїх контрактів.

- **Простота використання:** Незважаючи на свої складні аналітичні можливості, Mythril зберігає дружній інтерфейс, розроблений для спрощення процесу оцінки безпеки. Інструмент пропонує простий командний рядок і графічний інтерфейс користувача (GUI), що робить його доступним для розробників усіх рівнів кваліфікації. Завдяки інтуїтивно зрозумілим командам та інформативним результатам, Mythril сприяє безперешкодній інтеграції в робочий процес розробки, дозволяючи розробникам легко і впевнено проводити оцінку безпеки.
- **Активна підтримка спільноти:** Mythril користується підтримкою активної спільноти розробників, дослідників безпеки та ентузіастів блокчейну, які прагнуть підвищити рівень безпеки в екосистемі Ethereum. Ця співпраця, керована спільнотою, сприяє постійному вдосконаленню та інноваціям, а її учасники активно вдосконалюють і розширюють можливості Mythril для протидії новим загрозам і викликам безпеці, що постійно розвиваються. Крім того, відкритий характер Mythril сприяє прозорості та підзвітності, що дозволяє розробникам робити свій внесок у його розвиток і налаштовувати його функціональність відповідно до своїх конкретних потреб.
- **Інтеграція з конвеєрами розробки:** Mythril легко інтегрується з популярними конвеєрами розробки та робочими процесами безперервної інтеграції (CI), забезпечуючи автоматизований аналіз безпеки як частину життєвого циклу розробки програмного забезпечення. Інтегруючи Mythril в конвеєри CI/CD, розробники можуть проактивно виявляти та зменшувати вразливості на ранніх стадіях процесу розробки, мінімізуючи ризики безпеки та забезпечуючи надійність смарт-контрактів до їх розгортання.

Slither: Фреймворк статичного аналізу, який може виявити різні проблеми безпеки в коді Solidity, включаючи неправильну перевірку вхідних даних, небезпечні арифметичні операції та відсутні кваліфікатори функцій [24].

Slither - це потужний фреймворк статичного аналізу, ретельно розроблений для виявлення безлічі проблем безпеки, що ховаються в кодових базах Solidity. Завдяки

своїм складним можливостям аналізу, Slither відмінно справляється з виявленням критичних вразливостей, які загрожують безпеці і надійності смарт-контрактів. Пропонуємо вам ознайомитися з ключовими особливостями та значенням Slither:

- **Комплексне виявлення вразливостей:** Slither використовує передові методи статичного аналізу для виявлення широкого спектру проблем безпеки в коді Solidity. Від некоректної перевірки вхідних даних і небезпечних арифметичних операцій до відсутніх кваліфікаторів функцій і не тільки, Slither не залишає каменя на камені в своєму прагненні виявити вразливості, які можуть поставити під загрозу цілісність смарт-контрактів. Систематично аналізуючи код контракту і ретельно вивчаючи потенційні вектори атак, Slither дає розробникам можливість завчасно усунути слабкі місця в системі безпеки і захистити свої контракти від зловмисників.
- **Орієнтований на специфічні для Solidity вразливості:** На відміну від загальних інструментів статичного аналізу, Slither спеціально створений для вирішення унікальних проблем безпеки, притаманних розробці смарт-контрактів Solidity. Спеціалізуючись на специфічних для Solidity вразливостях, Slither пропонує індивідуальні висновки та рекомендації, пристосовані до тонкощів розробки смарт-контрактів Ethereum. Такий фокус на конкретному домені дозволяє Slither надавати цілеспрямовані вказівки та практичні рекомендації, які знаходять відгук у розробників блокчейну, підвищуючи ефективність оцінок безпеки та усунення вразливостей.
- **Зручний інтерфейс:** Slither надає пріоритет зручності та доступності, пропонуючи розробникам інтуїтивно зрозумілий та зручний інтерфейс для проведення аналізу безпеки. Інструмент надає простий командний рядок і графічний інтерфейс користувача (GUI), що робить його доступним для розробників усіх рівнів кваліфікації. Завдяки інформативним результатам та практичним висновкам Slither спрощує процес оцінки безпеки, дозволяючи розробникам виявляти та усувати вразливості з упевненістю та ефективністю.
- **Інтеграція з робочими процесами розробки:** Slither легко інтегрується з популярними конвеєрами розробки та робочими процесами безперервної

інтеграції (CI), полегшуючи автоматизований аналіз безпеки як невід'ємну частину життєвого циклу розробки програмного забезпечення. Інтегруючи Slither в CI/CD пайплайни, розробники можуть автоматизувати виявлення проблем безпеки і забезпечити усунення вразливостей на ранніх стадіях процесу розробки. Такий проактивний підхід до безпеки підвищує якість коду, мінімізує ризики та сприяє розвитку культури безпеки в командах розробників.

- **Постійне вдосконалення та підтримка спільноти:** Slither отримує вигоду від активної підтримки спільноти та постійного вдосконалення, завдяки відданій команді розробників та дослідників безпеки, які прагнуть розширити його можливості. Співпраця, керована спільнотою, сприяє інноваціям і вдосконаленню, а учасники активно вдосконалюють методи аналізу Slither і розширюють його можливості виявлення вразливостей. Крім того, відкритий характер Slither заохочує прозорість і співпрацю, що дає можливість розробникам робити свій внесок у його розвиток і адаптувати його функціональність під свої конкретні потреби.

Oyente: Інструмент, призначений для виявлення поширених вразливостей безпеки в смарт-контрактах Ethereum, таких як повторний вхід, неправильна обробка винятків і проблеми з порядком транзакцій [25].

Oyente є ключовим інструментом у сфері безпеки смарт-контрактів Ethereum, ретельно розробленим для виявлення безлічі поширених вразливостей, які становлять загрозу для цілісності та надійності смарт-контрактів. Завдяки своїм передовим методам аналізу та спеціалізованій увазі до специфічних питань безпеки Ethereum, Oyente надає розробникам можливість захистити свої контракти від експлуатації та захистити інтереси зацікавлених сторін. Ось всебічний огляд ключових особливостей та значення Oyente:

- **Спеціалізоване виявлення вразливостей:** Oyente спеціалізується на виявленні широкого спектру вразливостей безпеки, які поширені в смарт-контрактах Ethereum. Від атак повторного входу та неправильної обробки винятків до проблем з порядком транзакцій і не тільки, Oyente ретельно вивчає код контрактів, щоб виявити вразливості, які потенційно можуть поставити під

загрозу безпеку та стабільність блокчейн-додатків. Надаючи розробникам інформацію про поширені вектори атак та слабкі місця, які можна використати, Oyente сприяє проактивному усуненню вразливостей та зменшенню ризиків.

- **Розроблено для смарт-контрактів Ethereum:** На відміну від загальних інструментів аналізу безпеки, Oyente спеціально розроблений для розробки смарт-контрактів Ethereum, зосереджуючись на вразливостях, які є унікальними для екосистеми блокчейну Ethereum. Цей спеціалізований фокус дозволяє Oyente надавати цільові вказівки та практичні рекомендації, які знаходять відгук у розробників Ethereum, підвищуючи ефективність оцінки безпеки та усунення вразливостей. Вирішуючи специфічні для Ethereum проблеми безпеки, Oyente надає розробникам знання та інструменти, необхідні для створення стійких і безпечних смарт-контрактів.
- **Передові методи аналізу:** Oyente використовує передові методи статичного аналізу і символічного виконання для дослідження поведінки смарт-контрактів при різних сценаріях виконання. Систематично аналізуючи код контракту і моделюючи шляхи виконання транзакцій, Oyente виявляє потенційні вразливості і визначає проблемні області, які потребують подальшого вивчення. Ця сувора методологія аналізу дозволяє Oyente надавати вичерпну інформацію про стан безпеки смарт-контрактів, надаючи розробникам можливість приймати обґрунтовані рішення та ефективно визначати пріоритети щодо покращення безпеки.
- **Зручний інтерфейс:** Oyente пропонує зручний інтерфейс, який спрощує процес проведення аналізу безпеки для смарт-контрактів Ethereum. Інструмент надає інтуїтивно зрозумілий командний рядок і графічний інтерфейс користувача (GUI), що робить його доступним для розробників всіх рівнів кваліфікації. Завдяки інформативним результатам і практичним висновкам, Oyente спрощує процес оцінки безпеки, дозволяючи розробникам виявляти і усувати вразливості з упевненістю і ефективністю. Крім того, документація та ресурси підтримки Oyente полегшують передачу знань і розвиток навичок,

дозволяючи розробникам ефективно використовувати всі можливості інструменту.

- **Розробка під керівництвом спільноти:** Oyente отримує вигоду від активної підтримки спільноти та постійного вдосконалення, завдяки відданій команді розробників та дослідників безпеки, які прагнуть розширити його можливості. Співпраця, керована спільнотою, сприяє інноваціям і вдосконаленню, а учасники активно вдосконалюють методи аналізу Oyente і розширюють його можливості виявлення вразливостей. Крім того, відкритий характер Oyente заохочує прозорість і співпрацю, дозволяючи розробникам робити свій внесок у його розвиток і налаштовувати його функціональність відповідно до своїх конкретних потреб.

Використання інструментів статичного аналізу, таких як Mythril, Slither та Oyente, дозволяє розробникам виявляти та виправляти потенційні вразливості на ранніх стадіях життєвого циклу розробки. Ці інструменти ретельно перевіряють кодову базу на наявність поширених пасток безпеки, включаючи уразливості повторного входу, неперевірені виклики зовнішніх функцій та цілочисельні переповнення. Інтегруючи інструменти статичного аналізу в робочий процес розробки, розробники можуть виявляти та усувати вразливості проактивно, зменшуючи ризики ще до розгортання.

Інструменти формальної верифікації:

- **SmartCheck:** Інструмент формальної верифікації, розроблений Tezos Foundation, який допомагає перевіряти коректність і безпеку смарт-контрактів, написаних на різних мовах, включаючи Solidity [26].
- **VeriSolid:** фреймворк формальної верифікації для контрактів Solidity, який використовує модульні та вирішувані методи формальної верифікації для підтвердження функціональної коректності та інваріантних властивостей [27].

Для контрактів на великі суми, що управляють значними активами або здійснюють критичні транзакції, використання формальних інструментів верифікації, таких як SmartCheck та VeriSolid, забезпечує додатковий рівень безпеки. Методи формальної верифікації надають математичні докази правильності, пропонуючи

підвищену впевненість у безпеці та надійності контракту. Хоча формальна верифікація є більш суворим і ресурсоємним процесом, вона є незамінною для контрактів, де правильність має першорядне значення. Піддаючи критично важливі контракти формальній перевірці, розробники можуть обґрунтувати свої вимоги до безпеки неспростовними доказами, зміцнюючи довіру та впевненість у цілісності контракту.

Використовуючи ці інструменти безпеки, розробники можуть значно підвищити безпеку та надійність своїх смарт-контрактів Ethereum:

- Інтегруйте інструменти статичного аналізу в процес розробки, щоб виявити і виправити потенційні вразливості якомога раніше.
- Для контрактів з високою вартістю, які управляють значними активами або виконують критичні операції, розгляньте можливість використання інструментів формальної верифікації, щоб надати математичні докази правильності і підвищити впевненість в безпеці контракту.
- Регулярно переглядайте та оновлюйте контракт для усунення нових вразливостей та впровадження вдосконалених інструментів і методів аналізу безпеки.

Безпека - це безперервний процес, що вимагає регулярного перегляду та оновлення для усунення нових вразливостей і загроз, що з'являються. Розробники повинні проактивно відстежувати рекомендації з безпеки та впроваджувати вдосконалені інструменти та методи аналізу безпеки у свої процеси розробки. Регулярний перегляд коду, тестування на проникнення та оцінка вразливостей допомагають підтримувати стійкість смарт-контрактів до нових векторів атак. Залишаючись в курсі останніх подій у сфері безпеки та оперативно усуваючи вразливості, розробники можуть посилити захист своїх контрактів і забезпечити їхню довгострокову надійність.

Використовуючи ці інструменти та методи, розробники можуть створювати більш стійкі та надійні смарт-контракти Ethereum, здатні протистояти потенційним атакам та вразливостям. Проактивні заходи безпеки в поєднанні з постійною

пильністю та вдосконаленням мають важливе значення для захисту цілісності смарт-контрактів і зміцнення довіри в екосистемі Ethereum.

Впровадження цих інструментів і методів в процес розробки призведе до створення більш безпечних і надійних смарт-контрактів Ethereum, які зможуть краще протистояти потенційним атакам і вразливостям.

2.5 Можливість модернізації та оптимізація газу

Щоб забезпечити довгостроковий успіх смарт-контрактів Ethereum, розробники повинні враховувати як можливість оновлення, так і оптимізацію газу.

Контракти з можливістю оновлення та проксі-патерни:

- **Реалізуйте механізми оновлення:** Розробляйте контракти з можливістю оновлення, використовуючи такі методи, як контракти з можливістю оновлення та проксі-патерни. Це дозволить в майбутньому оновлювати та виправляти помилки без розгортання нової адреси контракту.
- **Відокремлюйте логіку контрактів та їх зберігання:** Використовуйте шаблон "вічного зберігання", щоб відокремити логіку контрактів від зберігання даних, що полегшить оновлення без втрати збереженої інформації.
- **Використовуйте проксі-контракти OpenZeppelin:** Використовуйте інфраструктуру контрактів OpenZeppelin, яку можна модернізувати, в тому числі шаблон "Прозорий проксі", для безпечного і усталеного підходу до оновлення контрактів.

Поєднання безпеки з оптимізацією використання газу:

- **Мінімізація використання сховищ:** Оптимізуйте використання сховища, використовуючи ефективні структури даних і уникаючи непотрібних змінних, щоб зменшити витрати на розгортання і транзакції.
- **Використовуйте газоефективні опкоди:** Вибирайте газоефективні опкоди Solidity, щоб мінімізувати споживання газу для виконання контрактів.

Оптимізуйте зовнішні виклики:

- Мінімізуйте зовнішні виклики контрактів і використовуйте методи пакетної обробки, щоб зменшити кількість транзакцій, необхідних для складних операцій.
- **Розробляйте модульні контракти:** Розробляйте модульні контракти, які можна компонувати для створення більш складних додатків, що дозволяють оптимізувати використання газу, розгортаючи лише необхідний функціонал.

Впроваджуючи безпечні механізми оновлення та балансуючи між безпекою та оптимізацією використання газу, розробники можуть створювати смарт-контракти Ethereum, які можна адаптувати, які є економічно ефективними та безпечними. Регулярний перегляд та вдосконалення стратегій оптимізації використання газу гарантує, що контракти залишатимуться доступними та ефективними в міру розвитку мережі Ethereum.

2.6 Стежте за змінами в Ethereum

Оскільки екосистема Ethereum продовжує розвиватися, розробники повинні бути в курсі змін, які можуть вплинути на безпеку і сумісність їх смарт-контрактів.

Бути в курсі оновлень протоколу Ethereum:

- **Слідкуйте за пропозиціями щодо вдосконалення Ethereum (EIP):** Будьте в курсі запропонованих змін до протоколу Ethereum, слідкуючи за EIP і беручи участь в обговореннях спільноти.
- **Підпишіться на джерела новин Ethereum:** Підпишіться на джерела новин, форуми та акаунти в соціальних мережах, присвячені Ethereum, щоб бути в курсі майбутніх оновлень протоколу, розробок в області безпеки і кращих практик.

Тестування та оновлення контрактів після зміни протоколу:

- **Тестуйте контракти на тестових мережах:** Перш ніж розгортати смарт-контракти в основній мережі, протестуйте їх на тестових мережах Ethereum, які впровадили останні оновлення протоколу, щоб переконатися в сумісності і виявити потенційні проблеми.

- **Слідкуйте за поведінкою контрактів:** Регулярно відстежуйте поведінку розгорнутих контрактів після оновлення протоколу, щоб виявити і усунути будь-які несподівані зміни або вразливості.
- **Проводьте аудит безпеки:** Регулярно проводить аудит безпеки смарт-контрактів, особливо після великих оновлень протоколу, щоб переконатися, що вони залишаються безпечними і відповідають останнім передовим практикам.

Залишаючись в курсі оновлень Ethereum і активно тестуючи, контролюючі і оновлюючи смарт-контракти, розробники можуть гарантувати, що їх додатки залишаться сумісними, безпечними і актуальними в екосистемі Ethereum, що швидко розвивається.

2.7 Ресурси спільноти для безпеки Ethereum

Спільнота Ethereum пропонує безліч ресурсів, які допоможуть розробникам підвищити безпеку своїх смарт-контрактів та бути в курсі найкращих практик.

Основні ресурси:

- “Кращі практики безпеки Ethereum” Вичерпний посібник з найкращими практиками для безпечної розробки смарт-контрактів на Ethereum, що охоплює такі теми, як якість коду, архітектура контракту та запобігання вразливостям [27].
- Звіти Consensys Diligence Reports: Оцінки безпеки смарт-контрактів і блокчейн-проектів на Ethereum, що надають цінну інформацію та рекомендації для розробників[28].

Участь у форумах для розробників та програмах винагород за виправлення помилок:

- Ethereum Stack Exchange: Форум питань і відповідей, де розробники можуть задавати питання, пов'язані з розробкою Ethereum, включаючи теми безпеки, і отримувати на них відповіді [29].
- Форум спільноти Ethereum: Форум для обговорення тем, пов'язаних з Ethereum, включаючи безпеку, з іншими розробниками та членами спільноти.

- Програми винагороди за виправлення помилок: Беріть участь у програмах винагороди за виправлення помилок, які пропонують такі платформи, як ImmuneFi, HackerOne або Ethereum Foundation, щоб вчитися у експертів з безпеки і допомагати виявляти вразливості в смарт-контрактах і додатках Ethereum.

Використовуючи ці ресурси спільноти, розробники можуть покращити своє розуміння безпеки Ethereum, бути в курсі останніх передових практик і зробити свій внесок в постійні зусилля по забезпеченню безпеки екосистеми Ethereum.

2.8 Практичні рекомендації щодо підвищення безпеки смарт-контрактів

Розробники, аудитори та користувачі можуть використовувати вичерпні інструкції, щоб забезпечити безпечне та надійне впровадження смарт-контракту. Ось найкращі практики для кожної групи:

Розробники:

1. Методи безпечного кодування Solidity:

- Запровадити перевірку введення та санітарну обробку, щоб запобігти неочікуваній поведінці.
- Використовуйте безпечні математичні бібліотеки, щоб запобігти вразливості цілочисельного переповнення/недоповнення.
- Уникайте використання старих або експериментальних функцій і віддавайте перевагу добре перевіреним бібліотекам.

2. Безпечний контроль доступу:

- Впровадити відповідні механізми контролю доступу, щоб обмежити конфіденційні функції або дані.
- Використовуйте принцип найменших привілеїв і перевіряйте дозволи для критичних операцій.
- Будьте обережні, використовуючи складні моделі контролю доступу та переконайтеся, що вони пройшли ретельну перевірку.

3. Комплексне тестування:

•Розробіть і виконайте модульні тести для перевірки очікуваної поведінки смарт-контракту.

•Охоплює як звичайні, так і крайні випадки, включаючи граничне тестування та негативне тестування.

•Розгляньте можливість використання тестових фреймворків, таких як Truffle або Hardhat, і використовуйте інструменти автоматизованого тестування.

4. Огляд коду:

•Введіть ретельний аналіз коду досвідченими розробниками для виявлення вразливостей, логічних недоліків і помилок кодування.

•Використовуйте посібники зі стилем та встановлені найкращі практики, щоб забезпечити послідовність і читабельність.

•Використовуйте такі інструменти, як лінтери, щоб виявити типові помилки кодування та впроваджувати стандарти кодування.

5. Аудити безпеки:

•Залучайте незалежних аудиторів безпеки або фірми, що спеціалізуються на безпеці смарт-контрактів.

•Введіть всебічну перевірку коду, дизайну та архітектури контракту.

•Усуньте будь-які вразливі або проблеми перед розгортанням.

Аудитори:

1. Розуміть бізнес-контекст:

•Отримайте глибоке розуміння мети смарт-контракту, вимог і очікуваної поведінки.

•Розгляньте результативні вектори атаки і вимоги безпеки, конкретні для домену контракту.

2. Ретельний аналіз коду:

•Проаналізуйте вихідний код контракту за наявності вразливостей, логічних недоліків і помилок кодування.

•Зверніть увагу на критичні функції, засоби контролю доступу, перевірку введених даних і зовнішні відносини.

- Використовуйте інструменти статичного аналізу та формальні методи, щоб покращити процес перевірки.

3. Тестування безпеки:

- Виконайте комплексне тестування безпеки, включаючи сканування вразливостей, фаззінгу та символічне виконання.
- Досліджуйте різні шляхи виконання та вхідні дані, щоб побачити ваші проблеми.
- Перевірте відповідність найкращим практикам безпеки та стандартам кодування.

4. Документація та звітність:

- Задокументуйте виявлені вразливості, їхній вплив і рекомендовані стратегії пом'якшення.
- Дайте чіткі та практичні рекомендації щодо вирішення виявлених проблем.
- Ефективно забезпечити результати розробникам і цікавим сторонам.

Користувачі:

1. Надійні джерела:

- Перевірте репутацію та довіру до розробника чи організації смарт-контракту.
- Трастові контракти , які пройшли незалежний аудит безпеки або були широко перевірені .

2. Огляд коду:

- Якщо можливо, перегляньте вихідний код контракту або розмістіть звіти перевірених аудиторів.
- Оцінити логіку контракту, засоби контролю доступу та деякі ризики, перш ніж взаємодіяти з ним.

3. Введіть до мінімуму вплив:

- Взаємодіяти лише з добре встановленими та використовуйте широкованими контрактами.
- Обмежити суму коштів або активів, довірених одному контракту.

- Розгляньте можливість використання гаманців із декількома підписами або службою умовного депонування для транзакцій із значною сумою.

4. Будьте в курсі:

- Слідкуйте за оновленнями безпеки, звітами про помилки та новинами, пов'язаними з вразливими місцями смарт-контрактів.
- Слідкуйте за довіреними експертами з безпеки та спільнотами, щоб бути в курсі передових методів і нових загроз.

Без цих вказівок, в основному важливо мати стратегії мінімізації ризиків і реагування на деякі порушення безпеки:

1. Зменшення ризиків:

- Регулярно оновлюйте контракти патчами безпеки та виправленнями помилок.
- Запровадити багатофакторну автентифікацію та надійний контроль доступу для адміністраторів контрактів.
- Відстежуйте контрактну діяльність і виявляйте аномалії або відчуття поведінки.

2. Реагування на інцидент:

- Розробити чітко визначений план реагування на інцидент, у якому описано кроки, які необхідно вжити у разі порушення безпеки.
- Дійте знову, щоб пом'якшити вплив і запобігти подальшому пошкодженню.
- Прозоро спілкуйтеся з постраждалими користувачами та зацікавленими сторонами.

3. Постійне вдосконалення:

- Вчіться на інцидентах безпеки та проводити аналізи після інцидентів, щоб застосувати сферу, які потребують покращення.
- Будьте в курсі останніх розробок у сфері безпеки смарт-контрактів і відповідним чином адаптувати методи безпеки.

Дотримуючись цих інструкцій і впроваджуючи надійні заходи безпеки, розробники, аудиторії та користувачі можуть зробити внесок у безпечне та надійне впровадження смарт-контрактів і мінімізувати деякі ризики.

Висновки за розділом 2

У цьому розділі ми докладно розглянули низку важливих аспектів роботи з платформою Ethereum, що є однією з найбільш популярних інфраструктур для розробки блокчейн-додатків. Ми розглянули роботу віртуальної машини Ethereum (EVM) і розуміння її функціонування, що є важливим для розробки ефективних смарт-контрактів. Далі ми зосередилися на безпеці мови програмування Solidity, яка використовується для написання смарт-контрактів, та висвітлили стратегії та кращі практики для забезпечення безпеки та уникнення вразливостей.

Наш висновок також охоплює важливі аспекти взаємодії з сервісами Ethereum, такими як використання газу, оптимізація та модернізація транзакцій. Ми також надали огляд оновлень Ethereum, щоб читачі могли залишатися в курсі останніх змін та вдосконалень на платформі. Крім того, ми вказали на ресурси спільноти, які можуть бути корисними для підвищення безпеки та знаходження відповідей на запитання.

Наші практичні рекомендації стануть цінним додатком до знань, які ми надали, допомагаючи розробникам і аудиторам забезпечити найвищий рівень безпеки та надійності їх смарт-контрактів.

РОЗДІЛ 3
НАПИСАННЯ СКАНЕРУ СМАРТ-КОНТРАКТІВ

Усвідомивши, наскільки важливою є безпека смарт-контрактів, ми почали працювати над створили сканер вразливостей SWC-100-136. Наша стратегія полягала в ретельному вивченні всіх змінних, необхідних для створення сканера смарт-контрактів, який би добре працював. Теоретично проаналізувавши вразливості в SWC-100-136, ми ретельно спланували і створили сканер. Потім було проведено обширне тестування, щоб підтвердити його ефективність і функціональність.

3.1 Аналіз важливих аспектів для написання сканеру смарт-контракту

Оскільки існує велика кількість мов програмування, вибір відповідної мови програмування для правдоподібно реалізації алгоритмів залишається складним питанням [30].

При написанні сканера для виявлення вразливостей в смарт-контрактах, вибір відповідної мови програмування як для сканера, так і для самих смарт-контрактів має вирішальне значення. Це рішення повинно ґрунтуватися на різних факторах, таких як специфічні сильні сторони кожної мови, досвід розробника та доступні ресурси.

Мови програмування для написання сканера

C - це базова мова, відома своїми низькорівневими можливостями, що надає розробникам прямий доступ до пам'яті та забезпечує високу швидкість виконання, що робить її оптимальним вибором для створення компонентів сканерів, критично важливих для продуктивності. Ось детальний огляд переваг та міркувань щодо мови C:

- **Прямий доступ до пам'яті:** Низькорівнева природа C надає розробникам безпрецедентний контроль над керуванням пам'яттю, дозволяючи безпосередньо маніпулювати адресами пам'яті та структурами даних. Такий прямий доступ до пам'яті дозволяє розробникам реалізовувати високоефективні

алгоритми та структури даних, необхідні для оптимізації продуктивності компонентів сканера, що працюють в умовах обмежених ресурсів.

- **Висока швидкість виконання:** Ефективний процес компіляції та мінімальні накладні витрати під час виконання сприяють тому, що мова C має репутацію мови з блискавично високою швидкістю виконання. Ця перевага особливо важлива для компонентів сканерів, які обробляють великі обсяги даних або проводять складні аналізи в режимі реального часу. Використовуючи швидкість і ефективність C, розробники можуть гарантувати, що операції сканера виконуються швидко і оперативно, підвищуючи загальну продуктивність системи і зручність роботи користувача.
- **Придатність для критичних до продуктивності компонентів:** Завдяки своїй здатності забезпечувати швидке та ефективне виконання, мова C добре підходить для реалізації критично важливих для продуктивності компонентів у сканерах смарт-контрактів. Такі завдання, як аналіз байт-коду, виявлення вразливостей і зіставлення шаблонів, вимагають високопродуктивних алгоритмів і структур даних для ефективної обробки великих кодових баз. Низькорівневі можливості C дозволяють розробникам оптимізувати ці компоненти для максимальної продуктивності, гарантуючи, що операції сканера будуть завершені вчасно і з мінімальними витратами ресурсів.

Однак важливо визнати, що використання C пов'язане з певними труднощами:

- **Статична типізація:** Система статичної типізації мови C вимагає від розробників оголошувати типи даних змінних під час компіляції, що може призвести до збільшення накладних витрат на розробку і потенційно перешкоджати гнучкості коду. Хоча статична типізація має такі переваги, як покращена зрозумілість коду та раннє виявлення помилок, вона може вимагати додаткових зусиль для керування складними структурами даних та пристосування до вимог, що змінюються.
- **Ручне керування пам'яттю:** Парадигма ручного керування пам'яттю у C покладає на розробників обов'язок явно виділяти та звільняти пам'ять, що збільшує ризик витоку пам'яті, переповнення буферів та інших вразливостей,

пов'язаних з пам'яттю. Хоча ручне керування пам'яттю надає розробникам детальний контроль над використанням пам'яті, воно вимагає пильної уваги до деталей та ретельного тестування, щоб забезпечити безпеку пам'яті та запобігти помилкам під час виконання.

C++, мова, похідна від C, спирається на фундамент свого попередника, додаючи потужні можливості об'єктно-орієнтованого програмування (ООП), що робить її особливо придатною для великомасштабних проектів. Ось детальний огляд переваг C++ над C:

- **Об'єктно-орієнтоване програмування:** C++ підтримує об'єктно-орієнтовану парадигму програмування, що дозволяє розробникам організовувати код у модульні, багаторазово використовувані компоненти, які називаються класами. Ця абстракція дозволяє створювати складні програмні системи, що складаються з взаємопов'язаних об'єктів, кожен з яких інкапсулює дані та поведінку. Полегшуючи організацію коду та абстрагування, C++ дає можливість розробникам ефективніше керувати складністю великомасштабних проектів, покращуючи зручність супроводу, масштабованість та повторне використання коду.
- **Покращене керування пам'яттю:** C++ пропонує кращі можливості керування пам'яттю порівняно з C, завдяки таким функціям, як конструктори, деструктори та інтелектуальні вказівники. Конструктори дозволяють автоматично ініціалізувати стан об'єкта, а деструктори полегшують очищення ресурсів після знищення об'єкта. Крім того, розумні вказівники, такі як std::unique_ptr та std::shared_ptr, забезпечують автоматичне керування пам'яттю і допомагають запобігти витокам пам'яті та "бовтанкам" вказівників. Використовуючи ці можливості, розробники можуть зменшити ризик помилок, пов'язаних з пам'яттю, та підвищити надійність своїх програм.
- **Розширені можливості мови:** На додаток до підтримки об'єктно-орієнтованого програмування, C++ надає багатий набір мовних можливостей, включаючи шаблони, винятки та стандартні бібліотеки. Шаблони уможливлюють узагальнене програмування, дозволяючи параметризувати

алгоритми та структури даних за типами. Винятки надають механізм для обробки помилок і виняткових ситуацій у структурований і послідовний спосіб, підвищуючи надійність коду і зручність його супроводу. Крім того, великі стандартні бібліотеки C++ пропонують широкий спектр готових функціональних можливостей для типових завдань, що скорочує час розробки та сприяє узгодженості коду.

- **Зворотна сумісність з C:** C++ підтримує зворотну сумісність з C, що дозволяє розробникам легко інтегрувати існуючі бази коду C у проекти на C++. Ця сумісність уможливлює поступову міграцію застарілого коду на C++, зберігаючи інвестиції в існуючі програмні ресурси та використовуючи переваги сучасних можливостей C++. Крім того, C++ пропонує механізми доступу до бібліотек C та виклику функцій C, що полегшує безперешкодну інтеграцію зі специфічними для платформи або сторонніми бібліотеками.

Python3 - мова програмування високого рівня, відома своєю читабельністю, простотою та універсальністю, що робить її надзвичайно придатною для створення прототипів сканерів та безперешкодної інтеграції існуючих бібліотек. Ось детальний огляд ключових атрибутів Python3 та міркувань щодо нього:

- **Читабельність і простота використання:** Python3 робить сильний акцент на читабельності та простоті, використовуючи чистий і лаконічний синтаксис, який має пріоритет над читабельністю для людини. Така філософія проектування робить Python3 особливо доступною для розробників усіх рівнів кваліфікації, сприяючи швидкій розробці та простоті розуміння. Інтуїтивно зрозумілий синтаксис мови та високорівневі абстракції дають змогу розробникам висловлювати складні ідеї у чіткий та стислий спосіб, сприяючи підвищенню продуктивності та співпраці в командах розробників.
- **Ідеально підходить для створення прототипів сканерів:** Простота використання Python3 та можливості швидкої розробки роблять її ідеальним вибором для створення прототипів сканерів смарт-контрактів. Динамічна типізація та інтерпретована природа мови дозволяють розробникам швидко ітерувати ідеї, експериментувати з різними алгоритмами та перевіряти

концепції дизайну з мінімальними накладними витратами. Велика стандартна бібліотека Python3 та багата екосистема сторонніх пакетів ще більше прискорюють процес розробки, надаючи готові рішення для поширених завдань, таких як файловий ввід/вивід, робота з мережею та маніпулювання даними.

- **Інтеграція з існуючими бібліотеками:** Python3 чудово інтегрується з існуючими бібліотеками та фреймворками завдяки потужній підтримці інтероперабельності та розширюваності. Розробники можуть використовувати можливості безшовної інтеграції Python3 для включення спеціалізованих бібліотек та інструментів у свої реалізації сканерів, розширюючи функціональність і підвищуючи продуктивність, не вигадуючи велосипед. Незалежно від того, чи взаємодіє вона з низькорівневими системними компонентами, чи використовує передові алгоритми машинного навчання, Python3 пропонує універсальну платформу для створення складних рішень для сканерів, пристосованих до конкретних вимог.
- **Міркування щодо продуктивності під час виконання:** Хоча Python3 пропонує неперевершену продуктивність і гнучкість для розробників, його інтерпретована природа може призвести до уповільнення часу виконання порівняно з такими скомпільованими мовами, як C та C++. Динамічна типізація та інтерпретація Python під час виконання створюють додаткові накладні витрати, які можуть вплинути на продуктивність компонентів сканера, що виконують обчислювально інтенсивні завдання. Однак продуктивність Python можна оптимізувати за допомогою таких методів, як профілювання коду, оптимізація алгоритмів і вибіркове використання скомпільованих розширень або бібліотек, орієнтованих на продуктивність.

Мови програмування для написання смарт-контрактів

Solidity є найкращим вибором для розробки смарт-контрактів на основі Ethereum, пропонуючи безліч переваг, які роблять її ідеальною мовою як для початківців, так і для досвідчених розробників. Ось розширений огляд переваг Solidity:

- **Велика спільнота розробників:** Solidity може похвалитися великою та активною спільнотою розробників, що складається з ентузіастів, експертів та професіоналів галузі. Ця розгалужена мережа сприяє співпраці, обміну знаннями та інноваціям, керованим спільнотою, надаючи розробникам безцінну підтримку та ресурси, які допомагають їм орієнтуватися в складнощах розробки смарт-контрактів.
- **Велика кількість навчальних ресурсів:** Solidity користується перевагами широкого спектру навчальних ресурсів, включаючи документацію, навчальні посібники, форуми та онлайн-курси. Ці ресурси розраховані на розробників будь-якого рівня кваліфікації, від початківців, які прагнуть осягнути основи, до досвідчених практиків, які заглиблюються в складні теми. Велика кількість навчальних матеріалів дозволяє розробникам-початківцям швидко ознайомитися з синтаксисом, можливостями та найкращими практиками Solidity, що прискорює їхній шлях до майстерності.
- **Відмінна інструментальна екосистема:** Екосистема Solidity рясніє безліччю високоякісних інструментів, призначених для спрощення розробки і тестування смарт-контрактів. Від інтегрованих середовищ розробки, таких як Remix і Visual Studio Code, до фреймворків для тестування, таких як Truffle і Ganache, розробники мають доступ до повного набору інструментів, які підвищують продуктивність, полегшують налагодження і забезпечують якість коду. Наявність надійного інструментарію спрощує процес розробки та дозволяє розробникам зосередитися на створенні інноваційних рішень, не обтяжуючи себе логістичними проблемами.
- **Зручний для початківців:** Інтуїтивно зрозумілий синтаксис Solidity та розгалужена інфраструктура підтримки роблять її надзвичайно зручним варіантом для розробників-початківців. Схожість мови зі звичними парадигмами програмування, такими як JavaScript та C++, знижує бар'єр для початківців, дозволяючи їм швидко засвоїти основні концепції та почати експериментувати з розробкою смарт-контрактів. Крім того, велика кількість навчальних ресурсів і форумів спільнот надають безцінні рекомендації та

допомогу, сприяючи розвитку і підвищенню кваліфікації розробників-початківців

Vyper є переконливою альтернативою Solidity, особливо для розробників Python, які шукають спрощений та безпечний підхід до розробки смарт-контрактів. Ось розширений огляд ключових переваг Vyper:

- **Простота і зрозумілість:** Vyper надає пріоритет простоті та читабельності, пропонуючи лаконічний та інтуїтивно зрозумілий синтаксис, який резонує з розробниками Python. Відмовляючись від складних функцій та надаючи перевагу ясності, Vyper покращує зрозумілість коду та мінімізує ймовірність помилок. Цей фокус на простоті сприяє більш доступному та зрозумілому досвіду розробки, особливо для розробників, які переходять від традиційної розробки програмного забезпечення до блокчейну.
- **Дизайн, орієнтований на аудит:** Філософія проектування Vyper обертається навколо полегшення надійного аудиту та формальних процесів верифікації. Обмежуючи набір функцій мови до підмножини Python, Vyper зводить до мінімуму неоднозначність і потенційні поверхні для атак, що полегшує аналіз і аудит смарт-контрактів. Такий підхід, орієнтований на аудит, вселяє впевненість у безпеці та надійності контрактів Vyper, дозволяючи розробникам закладати надійність у свої додатки з самого початку.
- **Можливість швидкого створення прототипів:** Спрощений набір функцій Vyper та синтаксис, подібний до Python, дозволяють швидко створювати прототипи смарт-контрактів. Розробники можуть швидко втілювати свої ідеї у виконуваний код, не стикаючись зі складнощами, притаманними іншим мовам. Така гнучкість прискорює цикл розробки, дозволяючи командам швидко ітерувати та експериментувати з різними дизайнами контрактів. Як результат, Vyper добре підходить для гнучких середовищ розробки, де швидкість і гнучкість мають першорядне значення.
- **Сприяє командній співпраці:** Чистий і зрозумілий синтаксис Vyper у поєднанні з його орієнтацією на читабельність сприяє безперешкодній співпраці в командах розробників. Єдність та передбачуваність мови полегшує

членам команди розуміння та перевірку коду один одного, сприяючи створенню спільного середовища, що сприяє колективному вирішенню проблем та обміну знаннями. Крім того, акцент Vyper на безпеці та аудиті підвищує впевненість команди в цілісності кодової бази, що ще більше посилює співпрацю та командну роботу.

Yul - це потужний інструмент в арсеналі розробників смарт-контрактів Ethereum, який пропонує прямий доступ до EVM і дозволяє тонко контролювати оптимізацію використання газу. Ось розширений огляд ключових функцій та переваг Yul:

- **Низькорівневий доступ до EVM:** Yul слугує низькорівневою мовою, яка надає розробникам безпрецедентний доступ до внутрішньої роботи EVM. Працюючи ближче до сирої EVM, розробники отримують кращий контроль над виконанням смарт-контрактів, що дозволяє точно оптимізувати використання газу та обчислювальну ефективність. Такий прямий доступ дозволяє розробникам адаптувати свої контракти до конкретних вимог до продуктивності та максимізувати використання ресурсів, що призводить до більш ефективного та економічно вигідного розгортання.
- **Оптимізація використання газу:** Однією з основних переваг Yul є її здатність оптимізувати використання газу в смарт-контрактах. Газ слугує паливом, яке живить транзакції та виконання контрактів в мережі Ethereum, а ефективне управління газом має вирішальне значення для мінімізації транзакційних витрат та максимізації масштабованості. Низькорівнева природа Yul дозволяє розробникам тонко налаштовувати контрактні операції, щоб мінімізувати споживання газу, тим самим оптимізуючи продуктивність і знижуючи комісію за транзакції. Використовуючи можливості Yul, розробники можуть створювати смарт-контракти, які є не тільки високоефективними, але й економічно вигідними для користувачів.
- **Високоефективні смарт-контракти:** Зосередженість Yul на оптимізації використання газу та низькорівневій взаємодії з EVM робить його особливо придатним для створення високоефективних смарт-контрактів. Працюючи

безпосередньо з EVM, розробники можуть усунути неефективність і накладні витрати, пов'язані з мовами більш високого рівня, в результаті чого код контракту стає більш компактним і впорядкованим. Ця ефективність особливо цінна в сценаріях, де обмеження ресурсів або проблеми масштабованості мають першорядне значення, наприклад, в додатках для децентралізованих фінансів або високочастотних торгових платформах. Yul дозволяє розробникам отримувати максимальну продуктивність від своїх контрактів, відкриваючи нові можливості для інновацій та оптимізації в блокчейні Ethereum.

Уважно вивчивши переваги і варіанти використання кожної мови програмування, розробники можуть приймати обґрунтовані рішення про найбільш підходящу мову для свого сканера смарт-контрактів і смарт-контрактів. Цей вибір в кінцевому підсумку вплине на ефективність, продуктивність і безпеку кінцевого продукту.

3.2 Теоретичний аналіз SWC-100-136

SWC-100: Функції, для яких не вказано тип видимості, за замовчуванням мають значення "public". Це може призвести до уразливості, якщо розробник забуде встановити видимість, і зловмисник може несанкціоновано або ненавмисно змінити стан функції [31].

Ця вразливість може призвести до

- Несанкціонованого доступу
- Втрати коштів
- Зміни правил контракту
- Заморожування або блокування контракту

Що є критичним для будь-якого смарт-контракту.

Рекомендується прийняти свідоме рішення про те, який тип видимості підходить для тієї чи іншої функції. External, public, internal, or private може бути використана для опису функцій виправлення. Визначення того, чи підходить тип

видимості для тієї чи іншої функції, слід робити свідомо. Таким чином, можна значно зменшити "поверхню атаки".

SWC-101: Переповнення/недоповнення відбувається, коли арифметична операція досягає максимального або мінімального розміру типу. Наприклад, якщо число зберігається у типі uint8, це означає, що воно зберігається у 8-бітному беззнаковому числу в діапазоні від 0 до 2^8-1. У комп'ютерному програмуванні цілочисельне переповнення виникає, коли арифметична операція намагається створити числове значення, яке виходить за межі діапазону, що може бути представлене заданою кількістю бітів - або більше за максимальне, або менше за мінімальне значення, що може бути представлене [32].

Ця вразливість може мати наступні наслідки:

- Некоректний обрахунок та обробка даних, що може призвести до неправильних результатів.
- Втрата коштів через неправильні фінансові операції.
- Порушення умов контракту через некоректні обчислення.
- Можливість виконання небажаних дій або атак на контракт через некоректне управління даними.

Що є критичним для будь-якого смарт-контракту.

Рекомендується у системі смарт-контрактів рекомендується постійно виконувати арифметичні операції з використанням перевірених безпечних математичних бібліотек.

SWC-102: Використання застарілої версії компілятора може бути проблематичним, особливо якщо в поточній версії компілятора є публічно оприлюднені баги та проблеми, які впливають на поточну версію компілятора [33].

Ця вразливість може призвести до:

- Серйозних вразливостей у безпеці
- Невідомих вад
- Сумісності зі сторонніми бібліотеками
- Загрози довірі користувачів

Що є критичним для будь-якого смарт-контракту.

Рекомендується використовувати найновішу версію компілятора Solidity.

SWC-103: Контракти слід розгортати тією самою версією компілятора і з тими самими прапорами, з якими вони були ретельно протестовані. Блокування прагматики допомагає гарантувати, що контракти не будуть випадково розгорнуті, наприклад, з використанням застарілої версії компілятора, яка може містити помилки, що негативно впливають на роботу контрактної системи [34].

Ця вразливість може призвести до:

- Серйозних синтаксичних помилок
- Потенційні загрози безпеці
- Несумісності з іншими контрактами

Що є критичним для будь-якого смарт-контракту.

Якщо контракт призначений для використання іншими розробниками, як у випадку з контрактами у бібліотеці або пакунку EthPM, прагматичні твердження можна залишити плаваючими. У протилежному випадку розробник повинен буде оновити прагматику вручну для того, щоб збирати локально.

SWC-104: Значення, що повертається при виклику повідомлення, не перевіряється. Виконання продовжується, навіть якщо викликаний контракт згенерував виключення. Якщо виклик випадково завершиться невдачею або зловмисник спричинило невдачу виклику, це може призвести до неочікуваної поведінки в подальшій логіці програми. [35].

Ця вразливість може призвести до:

- Непередбачуваного стану контракту
- Непередбачуваного споживання газу
- Проблеми з безпекою
- Недоступність послуг

Що є критичним для будь-якого смарт-контракту.

Якщо ви вирішите використовувати низькорівневі методи виклику, обов'язково перевірте значення, що повертається, на предмет того, що виклик може завершитися невдачею.

SWC-105: Через відсутність або недостатній контроль доступу зловмисники можуть вивести частину або весь Ефір з контрактного рахунку.

Ця помилка іноді виникає через ненавмисне розкриття функцій ініціалізації. Якщо неправильно назвати функцію, яка має бути конструктором, код конструктора потрапляє в байт-код часу виконання і може бути викликаний будь-ким для повторної ініціалізації контракту[36].

Що є критичним для будь-якого смарт-контракту.

Ця вразливість може призвести до:

- Втрати криптовалюти або токенів
- Порушення безпеки фінансових операцій
- Пошкодження репутації контракту або його розробника
- Загрози для безпеки користувачів, що використовують контракт

Якщо ви використовуєте низькорівневі методи виклику, обов'язково перевіряйте значення, що повертається, щоб врахувати можливість того, що виклик завершиться невдало.

SWC-106: Через відсутність або недостатній контроль доступу зловмисники можуть самознищити контракт [37].

Ця вразливість може призвести до:

- Повного або часткового видалення контракту
- Втрати контролю над ресурсами, що управляються контрактом
- Втрати доступу до функціональності, яка надавалася контрактом
- Порушення функціонування системи, побудованої на основі контракту

Що є критичним для будь-якого смарт-контракту.

Якщо функція самознищення не є абсолютно необхідною, подумайте про її видалення. Рекомендується використовувати метод мульти підписання, якщо є законний випадок використання, що вимагає схвалення від декількох сторін перед виконанням операції самознищення.

SWC-107: Однією з основних небезпек виклику зовнішніх контрактів є те, що вони можуть перехопити потік керування. В атаці зворотного входу (так званій атаці

рекурсивного виклику) зловмисний контракт повертається до викликаючого контракту до того, як завершиться перший виклик функції. Це може призвести до небажаної взаємодії різних викликів функції [38].

Ця вразливість може призвести до:

- Непередбачуваних змін стану контракту
- Втрати контролю над виконанням програми
- Втрати коштів через небажані операції, що викликаються рекурсивно

Що є критичним для будь-якого смарт-контракту.

Нижче наведено найкращі способи запобігти слабким місцям при повторному вході:

- Переконайтеся, що перед виконанням виклику всі внутрішні зміни стану були завершені. Тут це називається Checks-Effects-Interactions pattern.
- Використовуйте блокування повторного входу OpenZeppelin's ReentrancyGuard.

SWC-108: Явне позначення видимості полегшує виявлення невірних припущень щодо того, хто може отримати доступ до змінної [39].

Ця вразливість може призвести до:

- Несанкціонованого доступу до змінних
- Непередбачуваного стану програми через некоректну інтерпретацію видимості
- Втрати коштів через доступ до конфіденційної інформації
- Зниження безпеки контракту через неправильну інтерпретацію правил доступу

Що є критичним для будь-якого смарт-контракту.

Можна позначити змінні як public, internal, or private. Дайте кожній змінній стану чітке визначення видимості.

SWC-109: Неініціалізовані локальні змінні зберігання можуть вказувати на неочікувані місця зберігання в контракті, що може призвести до навмисних або не навмисних вразливостей [40].

Ця вразливість може призвести до:

- Несподіваної поведінки програми через некоректні дані
- Зміни стану контракту або його ресурсів
- Втрати контролю над виконанням програми через непередбачуваний стан
- Вразливостей, які можуть бути використані для атак на контракт

Що є критичним для будь-якого смарт-контракту.

Перевірте, чи передбачено в договорі об'єкт зберігання - часто це не так. Якщо достатньо локальної змінної, використовуйте атрибут memory, щоб чітко вказати місце зберігання змінної. Якщо потрібна змінна зберігання, ініціалізуйте її під час оголошення і додатково вкажіть місце зберігання storage.

SWC-110: Функція Solidity assert() призначена для ствердження інваріантів. Правильно функціонуючий код ніколи не повинен досягати помилкового твердження. Досяжне твердження може означати одну з двох речей:

- У контракті існує помилка, яка дозволяє йому перейти у недійсний стан;
- Оператор ствердження використовується некоректно, наприклад, для перевірки вхідних даних [41].

Що є критичним для будь-якого смарт-контракту.

Подумайте, чи умова assert(), яку було перевірено, дійсно є інваріантом. Якщо ні, використовуйте інструкцію require() замість інструкції assert().

Якщо причиною виключення є неочікувана поведінка коду, то виправте основну помилку або помилки, які дозволяють порушити твердження.

Ця вразливість може призвести до:

- Неправильного функціонування контракту через некоректне використання assert()
- Неочікуваних переходів контракту у недійсний стан
- Потенційної втрати коштів або ресурсів через неправильну обробку помилок.

SWC-111: Деякі функції та оператори у Solidity є застарілими. Їх використання призводить до зниження якості коду. У нових основних версіях компілятора Solidity

застарілі функції та оператори можуть призводити до побічних ефектів та помилок компіляції [42].

Ця вразливість може призвести до:

- Некоректної поведінки контракту через використання застарілих функцій та операторів
- Помилок компіляції у нових версіях компілятора Solidity
- Зниження ефективності та читабельності коду через застарілі конструкції

Що є критичним для будь-якого смарт-контракту.

Solidity пропонує замінники для застарілих структур. Оскільки більшість з них є псевдонімами, оновлення застарілих структур не змінить існуючу поведінку.

SWC-112: Існує спеціальний варіант виклику повідомлення, який називається delegatecall, який ідентичний виклику повідомлення, за винятком того, що код за цільовою адресою виконується в контексті викликаючого контракту, а msg.sender і msg.value не змінюють своїх значень. Це дозволяє смарт-контракту динамічно завантажувати код з іншої адреси під час виконання. Сховище, поточна адреса і баланс все ще відносяться до викликаючого контракту.

Виклик ненадійних контрактів дуже небезпечний, оскільки код за цільовою адресою може змінити будь-які значення сховища абонента і має повний контроль над балансом абонента [43].

Ця вразливість може призвести до:

- Несанкціонованого доступу до даних або зміни їх стану
- Втрати коштів через маніпуляцію з балансами або іншими значеннями сховища
- Зміни правил контракту або його стану через виконання ненадійного коду
- Загрози безпеці користувачів, оскільки недовірливий код може виконати шкідливі дії

Що є критичним для будь-якого смарт-контракту.

Важливо проявляти обережність при використанні "delegatecall" і ніколи не дзвонити в ненадійні контракти. Обов'язково порівнюйте цільову адресу, якщо вона отримана від користувача, з білим списком довірених контрактів.

SWC-113: Зовнішні виклики можуть випадково або навмисно зазнати невдачі, що може спричинити DoS-умову в контракті. Щоб мінімізувати шкоду від таких збоїв, краще ізолювати кожен зовнішній виклик в окрему транзакцію, яку може ініціювати одержувач виклику. Це особливо актуально для платежів, де краще дозволити користувачам знімати кошти, а не зараховувати їх автоматично (це також зменшує ймовірність проблем з лімітом газу) [44].

Ця вразливість може призвести до:

- DoS-атаки на контракт через невдачі зовнішніх викликів
- Втрати доступності контракту для користувачів через блокування виконання
- Потенційної втрати коштів або неправильної обробки операцій через невдачі в зовнішніх викликах

Що є критичним для будь-якого смарт-контракту.

Рекомендується дотримуватися найкращих практик роботи з викликами:

- Уникайте змішування декількох викликів в одній транзакції, особливо якщо виклики є частиною циклу.
- Завжди існує ймовірність того, що зовнішні дзвінки не будуть виконані.
- Застосовуйте логіку контракту для вирішення невдалих дзвінків.

SWC-114: Стан гонки, який найчастіше трапляється в мережі сьогодні, - це стан гонки в стандарті токенів ERC 20. Стандарт токенів ERC20 включає в себе функцію під назвою "затвердити", яка дозволяє адресі затверджувати іншій адресі витрачати токени від свого імені. Припустимо, що Аліса дозволила Єві витратити n своїх токенів, потім Аліса вирішує змінити дозвіл Єви на m токенів. Аліса надсилає виклик функції затвердження зі значенням n для Єви. Єва керує вузлом Ethereum, тому вона знає, що Аліса збирається змінити своє схвалення на m. Після цього Єва відправляє запит transferFrom, відправляючи n токенів Аліси собі, але вказуючи значно вищу ціну газу, ніж ціна транзакції Аліси. TransferFrom виконується першим, тому передає Єві n токенів і встановлює схвалення Єви на нуль. Потім виконується транзакція Аліси і встановлює схвалення Єви на m. Єва також відправляє ці m жетонів собі. Таким чином, Єва отримує n + m жетонів, хоча повинна була отримати не більше max(n,m). [45].

Що є критичним для будь-якого смарт-контракту.

Алгоритм розкриття хешу є одним з можливих рішень для вирішення проблеми расової дискримінації при наданні інформації в обмін на оплату. Замість того, щоб надавати відповідь, сторона, яка володіє відповіддю, надає хеш (сіль, адреса, відповідь) [сіль - число на свій вибір]; хеш і адреса відправника зберігаються в договорі. Потім відправник надсилає транзакцію з сіллю та відповіддю, щоб отримати приз. Контракт хешує вхідні дані (сіль, відправник повідомлення, відповідь), а потім порівнює отриманий хеш з хешем, який було збережено. Контракт видає приз, якщо хеш збігається.

Поле очікуваного поточного значення має бути додано до входів затвердження, і якщо поточний запас Єви відрізняється від очікуваного Алісою, затвердити має бути скасовано. Це найкращий спосіб вирішити проблему з умовами гонки ERC20. Однак в результаті ваш контракт більше не відповідає стандарту ERC20. Додайте функцію безпечного затвердження, якщо наявність контракту, сумісного з ERC20, має вирішальне значення для успіху вашого проекту. Перед тим, як вносити будь-які зміни в схвалення, користувач може звести їх до нуля, що допоможе пом'якшити умови перегонів ERC20.

SWC-115: tx.origin - це глобальна змінна в Solidity, яка повертає адресу акаунта, що відправив транзакцію. Використання цієї змінної для авторизації може зробити контракт вразливим, якщо авторизований акаунт викличе зловмисний контракт. Дзвінок може бути зроблений до вразливого контракту, який пройшов перевірку авторизації, оскільки tx.origin повертає початкового відправника транзакції, яким в даному випадку є авторизований акаунт [46].

Ця вразливість може призвести до:

- Несанкціонованого доступу до функціоналу через підроблені транзакції
- Втрати коштів або ресурсів через неправильну авторизацію на основі tx.origin
- Зміни правил контракту через виклик небезпечних дій з авторизованих акаунтів
- Потенційної втрати контролю над контрактом через вразливість авторизації на основі tx.origin

Що є критичним для будь-якого смарт-контракту.

Авторизація не повинна бути отримана за допомогою "tx.origin". Змініть його на "msg.sender".

SWC-116: У випадку з block.timestamp розробники часто намагаються використовувати його для запуску подій, що залежать від часу. Оскільки Ethereum децентралізований, вузли можуть синхронізувати час лише до певної міри. Більше того, зловмисники можуть змінювати мітку часу своїх блоків, особливо якщо це може дати їм переваги. Однак майнери не можуть встановлювати мітку часу меншу за попередню (інакше блок буде відхилено), а також не можуть встановлювати мітку часу надто далеко вперед у майбутньому. Беручи до уваги все вищесказане, розробники не можуть покладатися на точність наданої мітки часу.

Що стосується block.number, то з огляду на те, що час блоку в Ethereum зазвичай становить близько 14 секунд, можна передбачити часову дельту між блоками. Однак, час блоку не є постійним і може змінюватися з різних причин, наприклад, через реорганізацію форків і бомбу складності. Через мінливий час блоків, на block.number також не слід покладатися для точних розрахунків часу [47].

Що є критичним для будь-якого смарт-контракту.

Оскільки значення блоків неточні і можуть мати непередбачувані наслідки при використанні, розробники повинні створювати смарт-контракти з урахуванням цього. Або вони можуть використовувати оракули як заміну.

SWC-117: Реалізація системи криптографічного підпису в контрактах Ethereum часто передбачає, що підпис є унікальним, але підписи можуть бути змінені без володіння приватним ключем і все одно залишатися дійсними. Специфікація EVM визначає кілька так званих "попередньо скомпільованих" контрактів, одним з яких є ecrecover, що виконує відновлення відкритого ключа за еліптичною кривою. Зловмисник може трохи змінити три значення v, r і s, щоб створити інші дійсні підписи. Система, яка виконує перевірку підпису на рівні контракту, може бути вразливою до атак, якщо підпис є частиною хешу підписаного повідомлення. Дійсні підписи можуть бути створені зловмисником для відтворення раніше підписаних повідомлень [48].

Ця вразливість може призвести до:

- Несанкціонованого доступу до ресурсів або функціоналу контракту через використання підроблених підписів
- Втрати коштів або неправильної автентифікації користувачів через підроблені підписи
- Зміни правил контракту через підроблені підписи, що дозволяють виконання небажаних операцій
- Потенційної втрати довіри до системи через недостовірність аутентифікації за допомогою підписів.

Що є критичним для будь-якого смарт-контракту.

При визначенні того, чи оброблялися минулі повідомлення в рамках контракту, хеш підписаного повідомлення ніколи не повинен містити підпис.

SWC-118: Конструктори - це спеціальні функції, які викликаються лише один раз під час створення контракту. Вони часто виконують критичні, привілейовані дії, такі як встановлення власника контракту. До версії Solidity 0.4.22 єдиним способом визначення конструктора було створення функції з тим самим іменем, що й клас контракту, який його містить. Функція, призначена для створення конструктора, стає звичайною функцією, яку можна викликати, якщо її ім'я не збігається з іменем контракту. Така поведінка іноді призводить до проблем з безпекою, зокрема, коли код смарт-контракту повторно використовується з іншим ім'ям, але ім'я функції-конструктора не змінюється відповідним чином [49].

Ця вразливість може призвести до:

- Несанкціонованого доступу до контракту або його ресурсів.
- Втрати контролю над контрактом через неправильне встановлення власника або інших привілеїв.
- Зміни правил контракту через непередбачувані наслідки виклику конструктора.
- Заморожування або блокування контракту через некоректне створення.

Що є критичним для будь-якого смарт-контракту.

У версії 0.4.22 Solidity було додано нове ключове слово constructor для покращення чіткості визначень конструкторів. Тому рекомендується оновити ваш контракт до поточної версії компілятора Solidity і перейти на нове оголошення конструктора.

SWC-119: Цілісність дозволяє неоднозначне іменування змінних стану при у спадкуванні. Контракт А зі змінною x може успадкувати контракт В, в якому також визначена змінна стану x. Це призведе до появи двох окремих версій x, одна з яких буде доступна з контракту А, а інша - з контракту В. У більш складних контрактних системах ця умова може залишитися непоміченою і згодом призвести до проблем з безпекою.

Тінізація змінних стану також може відбуватися в межах одного контракту, коли є кілька визначень на рівні контракту та функції [50].

Ця вразливість може призвести до:

- Несанкціонованого доступу до даних.
- Втрати контролю над змінними стану.
- Зміни правил контракту без належного контролю.
- Заморожування або блокування контракту через конфлікти у визначенні стану.

Що є критичним для будь-якого смарт-контракту.

Уважно подивіться на розташування змінних сховища для ваших контрактних систем, щоб усунути будь-яку плутанину. Попередження компілятора - це гарне місце для початку, оскільки вони можуть виявити проблеми в одному контракті.

SWC-120: Здатність генерувати випадкові числа дуже корисна в різних сферах застосування. Одним з очевидних прикладів є азартні додатки, де для визначення переможця використовується генератор псевдовипадкових чисел. Однак створити достатньо сильне джерело випадковості в Ethereum дуже складно. Наприклад, використання block.timestamp є небезпечним, оскільки майнер може надати будь-яку мітку часу протягом декількох секунд і все одно його блок буде прийнятий іншими. Використання полів blockhash, block.difficulty та інших також є небезпечним, оскільки вони контролюються майнером. Якщо ставки високі, майнер може видобути

багато блоків за короткий час, орендуючи обладнання, вибрати блок, який має необхідний хеш для перемоги, і відкинути всі інші [51].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Несправедливості результатів у азартних або ігрових додатках.
- Викриття до атак підміни результатів генерації випадкових чисел.
- Втрати коштів або недостовірність генерованих даних.
- Потенційно непередбачувана поведінка контракту через несправедливість результатів.)

Що є критичним для будь-якого смарт-контракту.

- використання оракулів для доступу до зовнішніх випадкових джерел і криптографічна перевірка результату оракула в ланцюжку. наприклад, Chainlink VRF. Оскільки внутрішня частина системи відкине шахрайськи згенероване випадкове число, цей метод не залежить від віри в оракул.
- використання commitment scheme, наприклад, RANDAO.
- використання оракулів для доступу до зовнішніх випадкових джерел, наприклад, Oraclize Слід зазначити, що цей метод вимагає віри в оракул, тому використання більш ніж одного оракула може мати сенс.
- використовувати хеші блоків Bitcoin, оскільки їх видобуток коштує дорожче.

SWC-121: Іноді необхідно виконувати перевірку підпису в смарт-контрактах, щоб досягти кращої зручності використання або заощадити витрати на газ. Безпечна реалізація повинна захищати від атак повтору підпису, наприклад, відстежуючи всі оброблені хеші повідомлень і дозволяючи обробку лише нових хешів повідомлень. Зловмисник може атакувати контракт без такого контролю і отримати хеш повідомлення, надісланого іншим користувачем, оброблений кілька разів [52].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Атаки на повторне використання підписів, коли зловмисник використовує отриманий підпис для підтвердження транзакцій або інших дій без володіння приватним ключем.

- Несанкціоновані дії, такі як підписання чужих транзакцій або зміна стану контракту без дозволу власника.
- Втрати коштів або порушення безпеки змінних стану контракту через виконання невідомих або неправильних дій на підставі поддельного підпису.

Що є критичним для будь-якого смарт-контракту.

Щоб захиститися від атак з використанням відтворення підпису, візьміть до уваги наступні поради:

- Зберігайте хеш кожного повідомлення, яке обробив смарт-контракт. Коли надходять нові повідомлення, порівнюйте їх з тими, що вже існують, і виконуйте бізнес-логіку тільки в тому випадку, якщо хеш повідомлення новий.
- Надати адресу контракту для обробки повідомлення. Це гарантує, що повідомлення може бути використане лише в одному контракті.
- Ніколи не створюйте хеш повідомлення з включеним підписом. Гнучкість підпису впливає на функцію ecrecover.

SWC-122: Для систем смарт-контрактів характерно дозволяти користувачам підписувати повідомлення поза ланцюжком замість того, щоб безпосередньо просити їх здійснити транзакцію в ланцюжку, оскільки це забезпечує гнучкість і підвищену можливість передачі даних. Системи смарт-контрактів, які обробляють підписані повідомлення, повинні реалізовувати власну логіку для відновлення автентичності підписаних повідомлень перед їх подальшою обробкою. Обмеженням для таких систем є те, що смарт-контракти не можуть безпосередньо взаємодіяти з ними, оскільки вони не можуть підписувати повідомлення. Деякі реалізації перевірки підпису намагаються вирішити цю проблему, припускаючи дійсність підписаного повідомлення на основі інших методів, які не мають цього обмеження. Прикладом такого методу є використання msg.sender і припущення, що якщо підписане повідомлення походить з адреси відправника, то воно також було створене цією адресою. Це може призвести до вразливостей, особливо в сценаріях, де для передачі транзакцій можуть використовуватися проксі-сервери [53].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Можливість маніпулювання підписаними повідомленнями проміжною системою.
- Атаки на перехоплення або зміну повідомлень, які призводять до невірного виконання дій у смарт-контракті.
- Потенційні втрати коштів або інших негативних наслідків в результаті недостовірної обробки підписаних повідомлень.

Що є критичним для будь-якого смарт-контракту.

Не рекомендується використовувати альтернативні системи перевірки, які не вимагають адекватної перевірки підпису за допомогою ecrecover().

SWC-123: Конструкція require() у Solidity призначена для перевірки зовнішніх вхідних даних функції. У більшості випадків такі зовнішні вхідні дані надаються викликувачами, але вони також можуть бути повернуті функціями-викликувачами. У першому випадку ми називаємо їх порушеннями передумов. Порушення вимоги може вказувати на одну з двох можливих проблем:

Якщо ні, то в контракті, на підставі якого було надано зовнішні дані, має бути недолік, і було б добре змінити його код, щоб унеможливити надання неправильних даних.

- У договорі, який забезпечив зовнішні вхідні дані, існує помилка.
- Умова, яка використовується для вираження вимоги, є занадто сильною [54].

Що є критичним для будь-якого смарт-контракту.

Для того, щоб прийняти всі легітимні зовнішні вхідні дані, необхідну логічну вимогу слід знизити, якщо вона є надто суворою.

SWC-124: Дані смарт-контракту (наприклад, дані про власника контракту) постійно зберігаються в певному місці зберігання (наприклад, ключ або адреса) на рівні EVM. Контракт відповідає за те, щоб тільки авторизовані облікові записи користувачів або контрактів могли записувати дані в конфіденційні місця зберігання. Якщо зловмисник може писати в довільні місця зберігання контракту, перевірку авторизації можна легко обійти. Це може дозволити зловмиснику пошкодити сховище, наприклад, перезаписавши поле, в якому зберігається адреса власника контракту [55].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Порушення конфіденційності: Зловмисник може отримати доступ до конфіденційної інформації, яка зберігається в контракті, і модифікувати її відповідно до своїх потреб.
- Зміна важливих параметрів: Зловмисник може змінити важливі параметри контракту, такі як адреса власника, що може призвести до втрати контролю над контрактом або навіть до його зловживання.
- Пошкодження даних: Зловмисник може змінити або пошкодити дані, що зберігаються в контракті, що може призвести до втрати або зміни важливих інформаційних ресурсів.
- Заморожування або блокування контракту: Зловмисник може використовувати цю уразливість для заморожування або блокування роботи контракту, утримуючи або змінюючи важливі дані.

Що є критичним для будь-якого смарт-контракту.

Оскільки кожна структура даних має однаковий простір для зберігання (адресний простір), зазвичай рекомендується переконатися, що записи до однієї структури даних не призводять до ненавмисного перезапису записів в іншій структурі даних.

SWC-125: Solidity підтримує множинне успадкування, тобто один контракт може успадковувати декілька контрактів. Множинне успадкування призводить до неоднозначності, яка називається "діамантовою проблемою": якщо два або більше базових контрактів визначають одну і ту ж функцію, який з них слід викликати в дочірньому контракті? Solidity вирішує цю неоднозначність за допомогою зворотної лінеаризації C3, яка встановлює пріоритет між базовими контрактами.

Таким чином, базові контракти мають різні пріоритети, тому порядок успадкування має значення. Нехтування порядком успадкування може призвести до неочікуваної поведінки [56].

Що є критичним для будь-якого смарт-контракту.

Для розробників важливо правильно виражати успадкування у правильному порядку при успадкуванні декількох контрактів, особливо якщо вони містять

ідентичну функціональність. Як правило, контракти слід успадковувати від більш /general/ до більш /specific/.

SWC-126: Недостатня кількість атак на газ може бути здійснена на контракти, які приймають дані і використовують їх у субколл на іншому контракті. Якщо субколл не виконується, то або вся транзакція скасовується, або виконання продовжується. У випадку релейного контракту користувач, який виконує транзакцію, "експедитор", може ефективно цензурувати транзакції, використовуючи достатньо газу для виконання транзакції, але недостатньо для успішного виконання субкоманди [57].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Несанкціонований доступ: Зловмисник може скористатися цією вразливістю, щоб змінити стан смарт-контракту на свою користь.
- Втрата коштів: Користувачі можуть втратити кошти, якщо транзакція буде скасована або якщо субвиклик не виконає своїх зобов'язань.
- Зміна правил контракту: Зловмисник може скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту.
- Заморожування або блокування контракту: Смарт-контракт може бути заморожений або заблокований, якщо субвиклик не виконається.

Що є критичним для будь-якого смарт-контракту.

Існує два способи запобігти нестачі газу:

- Дозволити передачу транзакцій лише надійним користувачам.
- Переконайтеся, що експедитор має достатню кількість газу.

SWC-127: Solidity підтримує функціональні типи. Це означає, що змінній функціонального типу можна присвоїти посилання на функцію з відповідною сигнатурою. Функцію, збережену в такій змінній, можна викликати як звичайну функцію.

Проблема виникає тоді, коли користувач має можливість довільно змінювати змінну функціонального типу і таким чином виконувати інструкції випадкового коду. Оскільки Solidity не підтримує арифметику з вказівниками, змінити таку змінну на довільне значення неможливо. Однак, якщо розробник використовує інструкції

асемблера, такі як mstore або оператор присвоєння, в найгіршому випадку зловмисник може вказати змінну функціонального типу на будь-яку інструкцію коду, порушуючи необхідні перевірки та необхідні зміни стану [58].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Несанкціонований доступ: Зловмисник може скористатися цією вразливістю, щоб отримати доступ до приватних даних або функціональних можливостей смарт-контракту.
- Втрата коштів: Зловмисник може скористатися цією вразливістю, щоб викрасти кошти з смарт-контракту.
- Зміна правил контракту: Зловмисник може скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту на свою користь.
- Заморожування або блокування контракту: Зловмисник може скористатися цією вразливістю, щоб заморозити або заблокувати смарт-контракт.

Що є критичним для будь-якого смарт-контракту.

Збірка не повинна вимагати багато зусиль. Змінні функціонального типу не повинні мати довільних значень, які призначає їм користувач, на думку розробників.

SWC-128: Коли розгортаються смарт-контракти або викликаються функції всередині них, виконання цих дій завжди вимагає певної кількості газу, виходячи з того, скільки обчислень потрібно для їх завершення. Мережа Ethereum визначає ліміт газу в блоці, і сума всіх транзакцій, включених в блок, не може перевищувати цей поріг.

Шаблони програмування, які нешкідливі в централізованих додатках, можуть призвести до умов відмови в обслуговуванні в смарт-контрактах, коли вартість виконання функції перевищує ліміт газу блоку. Модифікація масиву невідомого розміру, який з часом збільшується, може призвести до такої умови відмови в обслуговуванні [59].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Відмова в обслуговуванні: Зловмисник може скористатися цією вразливістю, щоб заблокувати виконання транзакцій або запобігти доступу користувачів до смарт-контракту.

- Втрата коштів: Користувачі можуть втратити комісію за транзакцію, якщо транзакція не буде виконана через відмову в обслуговуванні.
- Зниження продуктивності: Відмова в обслуговуванні може призвести до зниження продуктивності мережі Ethereum.

Що є критичним для будь-якого смарт-контракту.

Якщо ви передбачаєте мати великі масиви, які розширюються з часом, дійте з обережністю. Уникайте будь-яких дій, які вимагають циклічного перегляду всієї структури даних.

Якщо у вас немає іншого вибору, окрім як виконати цикл над масивом невідомого розміру, ви повинні передбачити, що це може зайняти кілька блоків і, відповідно, багато транзакцій.

SWC-129: Друкарська помилка може виникнути, наприклад, коли метою визначеної операції є додавання числа до змінної (+=), але вона випадково була визначена неправильно (=+), що призвело до помилки, яка виявилася коректним оператором. Замість того, щоб обчислити суму, він знову ініціалізує змінну.

Унарний оператор + застарілий у нових версіях компілятора солідності [60].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Несанкціонований доступ: Зловмисник може скористатися цією вразливістю, щоб отримати доступ до приватних даних або функціональних можливостей смарт-контракту.
- Втрата коштів: Зловмисник може скористатися цією вразливістю, щоб викрасти кошти з смарт-контракту.
- Зміна правил контракту: Зловмисник може скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту на свою користь.
- Заморожування або блокування контракту: Зловмисник може скористатися цією вразливістю, щоб заморозити або заблокувати смарт-контракт.

Що є критичним для будь-якого смарт-контракту.

Одним із способів запобігти цій вразливості є перевірка передумов для всіх математичних операцій або використання надійної бібліотеки для арифметичних обчислень, наприклад, OpenZeppelin's SafeMath.

SWC-130: Зловмисники можуть використовувати символ юнікоду Right-To-Left-Override для примусового відображення RTL-тексту і вводити користувачів в оману щодо справжніх намірів контракту [61].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Несанкціонований доступ: Зловмисники можуть скористатися цією вразливістю, щоб отримати доступ до приватних даних або функціональних можливостей смарт-контракту.
- Втрата коштів: Зловмисники можуть скористатися цією вразливістю, щоб викрасти кошти з смарт-контракту.
- Зміна правил контракту: Зловмисники можуть скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту на свою користь.
- Заморожування або блокування контракту: Зловмисники можуть скористатися цією вразливістю, щоб заморозити або заблокувати смарт-контракт.

Що є критичним для будь-якого смарт-контракту.

Символ U+202E має дуже мало загальноприйнятих варіантів використання. Він не повинен бути присутнім у вихідному коді смарт-контракту.

SWC-131: Невикористовувані змінні дозволені у Solidity і не становлять безпосередньої загрози безпеці. Однак, найкращою практикою буде уникати їх, наскільки це можливо:

Що є критичним для будь-якого смарт-контракту.

- Видаліть усі зайві змінні з вихідного коду.
- спричинити збільшення обчислень (і непотрібне споживання газу)
- вказувати на помилки або неправильні структури даних, що, як правило, є ознакою низької якості коду
- спричиняють шум у коді та погіршують його читабельність [62]

SWC-132: Контракти можуть поводитися помилково, коли вони строго припускають певний баланс ефіру. Завжди можна примусово відправити ефір на контракт (без запуску його резервної функції), використовуючи самознищення або

майнінг на рахунок. У найгіршому випадку це може призвести до умов DOS, які можуть зробити контракт непридатним для використання [63].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Відмова в обслуговуванні: Зловмисники можуть скористатися цією вразливістю, щоб зробити смарт-контракт недоступним для використання.
- Втрата коштів: Зловмисники можуть скористатися цією вразливістю, щоб вкрасти кошти з смарт-контракту.
- Зміна правил контракту: Зловмисники можуть скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту на свою користь.

Що є критичним для будь-якого смарт-контракту.

Уникайте суворих перевірок на рівність щодо ефірного балансу контракту.

SWC-133: Використання abi.encodePacked() з декількома аргументами змінної довжини може у певних ситуаціях призвести до хеш-колізії. Оскільки abi.encodePacked() пакує всі елементи по порядку, незалежно від того, чи є вони частиною масиву, ви можете переміщати елементи між масивами і, доки всі елементи знаходяться у тому самому порядку, вона повертатиме однакове кодування. У ситуації з перевіркою підпису зловмисник може скористатися цим, змінивши позицію елементів у попередньому виклику функції, щоб ефективно обійти авторизацію [64].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Несанкціонований доступ: Зловмисники можуть скористатися цією вразливістю, щоб отримати доступ до приватних даних або функціональних можливостей смарт-контракту.
- Втрата коштів: Зловмисники можуть скористатися цією вразливістю, щоб викрасти кошти з смарт-контракту.
- Зміна правил контракту: Зловмисники можуть скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту на свою користь.

Що є критичним для будь-якого смарт-контракту.

Важливо переконатися, що abi.encodePacked() не може надати відповідну сигнатуру з різними параметрами. Для цього використовуйте масиви фіксованої

довжини або забороніть користувачам доступ до параметрів, що використовуються у abi.encodePacked(). Як альтернатива, ви можете просто використовувати abi.encode().

Також рекомендується використовувати захист від повторного відтворення (див. SWC-121), хоча зловмисник все одно може обійти його за допомогою запуску на випередження.

SWC-134: Функції transfer() і send() пересилають фіксовану кількість 2300 газів. Історично склалося так, що ці функції часто рекомендували використовувати для переказу вартості, щоб захиститися від атак на повторний вхід. Однак вартість газу в інструкціях EVM може значно змінитися під час хардфорків, що може порушити вже розгорнуті контрактні системи, які роблять фіксовані припущення про вартість газу. Наприклад. EIP 1884 зламав кілька існуючих смарт-контрактів через збільшення вартості інструкції SLOAD [65].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Відмова в обслуговуванні: Зловмисники можуть скористатися цією вразливістю, щоб зробити смарт-контракт недоступним для використання.
- Втрата коштів: Зловмисники можуть скористатися цією вразливістю, щоб вкрасти кошти з смарт-контракту.
- Зміна правил контракту: Зловмисники можуть скористатися цією вразливістю, щоб змінити правила роботи смарт-контракту на свою користь.

Що є критичним для будь-якого смарт-контракту.

При виконанні викликів утриматися від використання send() і transfer(), а також не передавайте фіксовану кількість газу в будь-який інший спосіб. Замість цього використовуйте use.call.value(...)(""). Щоб зупинити атаки на повторний вхід, використовуйте блокування повторного входу або патерн перевірки-ефекти-взаємодії.

SWC-135: У Solidity можна писати код, який не створює передбачуваних ефектів. Наразі компілятор Solidity не видає попередження для коду без ефектів. Це може призвести до появи "мертвого" коду, який не виконує належним чином заплановану дію.

Наприклад, легко пропустити кінцеві дужки у msg.sender.call.value(address(this).balance)("");, що може призвести до виконання функції без переказу коштів до msg.sender. Хоча цього слід уникати, перевіряючи значення, що повертається у виклику [66].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Неефективність: Мертвий код може зробити смарт-контракт менш ефективним, оскільки він витрачає ресурси без будь-якої користі.
- Вразливості: Мертвий код може бути використаний зловмисниками для приховування шкідливого коду або для обходу механізмів безпеки.
- Складність: Мертвий код може ускладнити розуміння та обслуговування смарт-контракту.

Що є критичним для будь-якого смарт-контракту.

Переконатися, що ваш контракт функціонує за призначенням, дуже важливо. Щоб переконатися, що код працює правильно, пишіть юніт-тести.

SWC-136: Поширеною помилкою є думка, що змінні приватного типу не можна прочитати. Навіть якщо ваш контракт не опублікований, зловмисники можуть переглянути транзакції контракту, щоб визначити значення, що зберігаються в стані контракту. З цієї причини важливо, щоб незашифровані приватні дані не зберігалися в коді або стані контракту [67].

Проблеми з випадковістю можуть призвести до наступних наслідків:

- Неочікуваного звернення керування
- Витік даних: Зловмисники можуть отримати доступ до приватної інформації, такої як особисті дані або фінансові дані.
- Зниження репутації: Витік даних може завдати шкоди репутації компанії або проекту, який розробив смарт-контракт.
- Втрата коштів: Зловмисники можуть використовувати витік даних для крадіжки коштів або інших цінних активів.

Що є критичним для будь-якого смарт-контракту.

Будь-яка конфіденційна інформація повинна бути надійно зашифрована або зберігатися поза ланцюжком.

3.3 Написання сканеру смарт-контракту на мові Python3

Після аналізу усіх вразливостей ми вирішили написати код. Частину коду ви бачите перед собою.

```
    def swc136(solidity_code):
        # Define pattern for unencrypted private data
        pattern = r"(private\s+)(.*?)\s*=\s*(.*?)\s*;"
        # Find all occurrences of unencrypted private data
        matches = re.findall(pattern, solidity_code)
        if matches:
            print("SWC-136 vulnerability discovered. Unencrypted private data found:")
            for match in matches:
                print(f"- {match[2]}")
            print("\nIt's important that unencrypted private data is not stored in the contract code or state.")
        else:
            print("No unencrypted private data found. No SWC-136")
def swc135(solidity_code):
    pattern = r"msg\.sender\.call\.value\(.*\)\s*\(\)\s*"
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Effect-free code found:")
        for match in matches:
            print(f"- {match}")
        print("\nIt's recommended to check the return value of the call to ensure the intended effect is being produced.")
    else:
        print("No effect-free code found.")
def swc134(solidity_code):
```

```
    pattern = r"(\._callable\.(transfer|send|call\.value|call))\((.*?)\)"
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("The transfer() and send() functions forward a fixed amount of 2300 gas.")
        print("Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks.")
        print("However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs.")
        print("For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.")
        return True
    return  print("False swc134")
```

У весь код ви можете побачити в додатку А.

Проведемо тестування тестового смарт контракту:

Тест 1:

```
pragma solidity 0.8.1;
interface ICallable {
    function callMe() external;
}
contract HardcodedNotGood {
    address payable _callable = 0xaAaAaAaaAaAaAaaAaAAAAAAAAAaaaAaAaAaaAaaAa;
    ICallable callable = ICallable(_callable);
    constructor() public payable {
    }
    function doTransfer(uint256 amount) public {
        _callable.transfer(amount);
    }
    function doSend(uint256 amount) public {
```

```
        _callable.send(amount);
    }
    function callLowLevel() public {
        _callable.call.value(0).gas(10000)("");
    }
    function callWithArgs() public {
        callable.callMe{gas: 10000}();
    }
}
```

Відповідь яку ми отримали:

SWC-100 vulnerability discovered. Potential function visibility vulnerabilities found:

- function callMe
- function doTransfer
- function doSend
- function callLowLevel
- function callWithArgs

No potential outdated compiler version SWC-102 vulnerabilities found.

No potential pragma version SWC-103 vulnerabilities found.

No potential unchecked message call SWC-104 vulnerabilities found.

No potential Ether withdrawal SWC-105 vulnerabilities found.

SWC-106 vulnerability discovered. No potential self-destruct vulnerabilities found.

No reentrancy SWC-107 vulnerability found.

SWC-108 vulnerability discovered.Variable visibility warnings:

Line 7: Variable payable does not have explicit visibility.

SWC-109 vulnerability discovered. Uninitialized local storage variable warnings:

Line 7: Uninitialized local storage variable: payable

The assert() function is not used in the Solidity code. No SWC-110

No deprecated functions or operators are used in the Solidity code. No SWC-111

The delegatecall() function is not used in the Solidity code. No SWC-112

SWC-113 vulnerability discovered.The following external calls are used:

- transfer

- send

- call.value

No arithmetic operations are used in the Solidity code. No SWC-114

The 'tx.origin' variable is not used in the Solidity code. No SWC-115.

The 'block.timestamp' and 'block.number' variables are not used in the Solidity code. No SWC-116.

The 'ecrecover' function is not used in the Solidity code. No SWC-117

A constructor with a name different from the contract name is not used in the Solidity code. No SWC-118.

No potential overflow/underflow SWC-119 vulnerability found.

No potential issues with random number generation found. No SWC-120.

No potential issues with signature verification found. No SWC-121.

No potential issues with signature verification found.SWC122

No potential issues with require() construct found. No SWC-123

No storage locations are used in the Solidity code. No SWC-124

No multiple inheritance is used in the Solidity code. No SWC-125

No 'call' or 'delegatecall' is used in the Solidity code. No SWC-126

No 'assembly' is used in the Solidity code. No SWC-127

No potential SWC-100 vulnerability gas limit issues found.

No potential SWC-129 vulnerability found.

False SWC-130

No unused SWC-131 vulnerability found.

No SWC-132

No SWC-133

False SWC-134

No effect-free code found. No SWC-135

No unencrypted private data found. No SWC-136

Тест 2:

pragma solidity ^0.4.22;

```
contract Rubixi {
    //Declare variables for storage critical to contract
    uint private balance = 0;
    uint private collectedFees = 0;
    uint private feePercent = 10;
    uint private pyramidMultiplier = 300;
    uint private payoutOrder = 0;

    address private creator;
    //Sets creator
    function DynamicPyramid() {
        creator = msg.sender;
    }
    modifier onlyowner {
        if (msg.sender == creator) _;
    }
    struct Participant {
        address etherAddress;
        uint payout;
    }
    Participant[] private participants;
    //Fallback function
    function() {
        init();
    }
    //init function run on fallback
    function init() private {
        //Ensures only tx with value of 1 ether or greater are processed and added to pyramid
        if (msg.value < 1 ether) {
```

```
                collectedFees += msg.value;
                return;
            }
            uint _fee = feePercent;
            //50% fee rebate on any ether value of 50 or greater
            if (msg.value >= 50 ether) _fee /= 2;
            addPayout(_fee);
        }

        //Function called for valid tx to the contract
        function addPayout(uint _fee) private {
            //Adds new address to participant array
            participants.push(Participant(msg.sender, (msg.value * pyramidMultiplier)
/ 100));
            //These statements ensure a quicker payout system to later pyramid entrants,
so the pyramid has a longer lifespan
            if (participants.length == 10) pyramidMultiplier = 200;
            else if (participants.length == 25) pyramidMultiplier = 150;
            // collect fees and update contract balance
            balance += (msg.value * (100 - _fee)) / 100;
            collectedFees += (msg.value * _fee) / 100;
            //Pays earlier participiants if balance sufficient
            while (balance > participants[payoutOrder].payout) {
                uint payoutToSend = participants[payoutOrder].payout;
                participants[payoutOrder].etherAddress.send(payoutToSend);
                balance -= participants[payoutOrder].payout;
                payoutOrder += 1;
            }
        }
        //Fee functions for creator
```

```
function collectAllFees() onlyowner {
        if (collectedFees == 0) throw;
        creator.send(collectedFees);
        collectedFees = 0;
}
function collectFeesInEther(uint _amt) onlyowner {
        _amt *= 1 ether;
        if (_amt > collectedFees) collectAllFees();

        if (collectedFees == 0) throw;
        creator.send(_amt);
        collectedFees -= _amt;
}
function collectPercentOfFees(uint _pcent) onlyowner {
        if (collectedFees == 0 || _pcent > 100) throw;
        uint feesToCollect = collectedFees / 100 * _pcent;
        creator.send(feesToCollect);
        collectedFees -= feesToCollect;
}
//Functions for changing variables related to the contract
function changeOwner(address _owner) onlyowner {
        creator = _owner;
}
function changeMultiplier(uint _mult) onlyowner {
        if (_mult > 300 || _mult < 120) throw;
        pyramidMultiplier = _mult;
}
function changeFeePercentage(uint _fee) onlyowner {
        if (_fee > 10) throw;
        feePercent = _fee;
```

```
    }
    //Functions to provide information to end-user using JSON interface or other interfaces
    function currentMultiplier() constant returns(uint multiplier, string info) {
        multiplier = pyramidMultiplier;
        info = 'This multiplier applies to you as soon as transaction is received, may be lowered to hasten payouts or increased if payouts are fast enough. Due to no float or decimals, multiplier is x100 for a fractional multiplier e.g. 250 is actually a 2.5x multiplier. Capped at 3x max and 1.2x min.';
    }
    function currentFeePercentage() constant returns(uint fee, string info) {
        fee = feePercent;
        info = 'Shown in % form. Fee is halved(50%) for amounts equal or greater than 50 ethers. (Fee may change, but is capped to a maximum of 10%)';
    }
    function currentPyramidBalanceApproximately() constant returns(uint pyramidBalance, string info) {
        pyramidBalance = balance / 1 ether;
        info = 'All balance values are measured in Ethers, note that due to no decimal placing, these values show up as integers only, within the contract itself you will get the exact decimal value you are supposed to';
    }
    function nextPayoutWhenPyramidBalanceTotalsApproximately() constant returns(uint balancePayout) {
        balancePayout = participants[payoutOrder].payout / 1 ether;
    }
    function feesSeperateFromBalanceApproximately() constant returns(uint fees) {
        fees = collectedFees / 1 ether;
    }
    function totalParticipants() constant returns(uint count) {
```

```
            count = participants.length;
        }
        function numberOfParticipantsWaitingForPayout() constant returns(uint count)
{
            count = participants.length - payoutOrder;
        }

        function participantDetails(uint orderInPyramid) constant returns(address
Address, uint Payout) {
            if (orderInPyramid <= participants.length) {
                Address = participants[orderInPyramid].etherAddress;
                Payout = participants[orderInPyramid].payout / 1 ether;
            }
        }
    }
```

Вивід сканеру:

SWC-100 vulnerability discovered. Potential function visibility vulnerabilities found:

- function DynamicPyramid
- function
    function
- function init
- function addPayout
- function collectAllFees
- function collectFeesInEther
- function collectPercentOfFees
- function changeOwner
- function changeMultiplier
- function changeFeePercentage
- function currentMultiplier
- function currentFeePercentage

- function currentPyramidBalanceApproximately
- function nextPayoutWhenPyramidBalanceTotalsApproximately
- function feesSeperateFromBalanceApproximately
- function totalParticipants
- function numberOfParticipantsWaitingForPayout
- function participantDetails

Functions that do not have a function visibility type specified are public by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes. To prevent this, it's a good idea to make a conscious decision on which visibility type is appropriate for a function. This can dramatically reduce the attack surface of a contract system.

No arithmetic overflow/underflow warnings found.

No potential outdated compiler version SWC-102 vulnerabilities found.

No potential pragma version SWC-103 vulnerabilities found.

No potential unchecked message call SWC-104 vulnerabilities found.

No potential Ether withdrawal SWC-105 vulnerabilities found.

SWC-106 vulnerability discovered. No potential self-destruct vulnerabilities found.

No reentrancy SWC-107 vulnerability found.

SWC-108 vulnerability discovered.Variable visibility warnings:

Line 24: Variable Participant does not have explicit visibility.

Line 25: Variable etherAddress does not have explicit visibility.

Line 26: Variable payout does not have explicit visibility.

Line 44: Variable _fee does not have explicit visibility.

Line 53: Variable to does not have explicit visibility.

Line 66: Variable payoutToSend does not have explicit visibility.

Line 82: Variable _amt does not have explicit visibility.

Line 92: Variable _pcent does not have explicit visibility.

Line 95: Variable feesToCollect does not have explicit visibility.

Line 101: Variable _owner does not have explicit visibility.

Line 105: Variable _mult does not have explicit visibility.

Line 111: Variable _fee does not have explicit visibility.

Line 118: Variable info does not have explicit visibility.

Line 123: Variable info does not have explicit visibility.

Line 128: Variable info does not have explicit visibility.

Line 133: Variable balancePayout does not have explicit visibility.

Line 137: Variable fees does not have explicit visibility.

Line 141: Variable count does not have explicit visibility.

Line 145: Variable count does not have explicit visibility.

Line 149: Variable Payout does not have explicit visibility.

SWC-109 vulnerability discovered. Uninitialized local storage variable warnings:

Line 7: Uninitialized local storage variable: private

Line 8: Uninitialized local storage variable: private

Line 9: Uninitialized local storage variable: private

Line 10: Uninitialized local storage variable: private

Line 11: Uninitialized local storage variable: private

Line 13: Uninitialized local storage variable: private

Line 24: Uninitialized local storage variable: Participant

Line 25: Uninitialized local storage variable: etherAddress

Line 26: Uninitialized local storage variable: payout

Line 52: Uninitialized local storage variable: _fee

Line 53: Uninitialized local storage variable: to

Line 82: Uninitialized local storage variable: _amt

Line 92: Uninitialized local storage variable: _pcent

Line 101: Uninitialized local storage variable: _owner

Line 105: Uninitialized local storage variable: _mult

Line 111: Uninitialized local storage variable: _fee

Line 118: Uninitialized local storage variable: info

Line 123: Uninitialized local storage variable: info

Line 128: Uninitialized local storage variable: info

Line 133: Uninitialized local storage variable: balancePayout

Line 137: Uninitialized local storage variable: fees

Line 141: Uninitialized local storage variable: count

Line 145: Uninitialized local storage variable: count

Line 149: Uninitialized local storage variable: Payout

The assert() function is not used in the Solidity code. No SWC-110

No deprecated functions or operators are used in the Solidity code. No SWC-111

The delegatecall() function is not used in the Solidity code. No SWC-112

SWC-113 vulnerability discovered.The following external calls are used:

- send

- send

- send

- send

No arithmetic operations are used in the Solidity code. No SWC-114

The 'tx.origin' variable is not used in the Solidity code. No SWC-115.

The 'block.timestamp' and 'block.number' variables are not used in the Solidity code. No SWC-116.

The 'ecrecover' function is not used in the Solidity code. No SWC-117

A constructor with a name different from the contract name is not used in the Solidity code. No SWC-118.

No potential overflow/underflow SWC-119 vulnerability found.

No potential issues with random number generation found. No SWC-120.

No potential issues with signature verification found. No SWC-121.

No potential issues with signature verification found.SWC122

No potential issues with require() construct found. No SWC-123

No storage locations are used in the Solidity code. No SWC-124

No multiple inheritance is used in the Solidity code. No SWC-125

No 'call' or 'delegatecall' is used in the Solidity code. No SWC-126

No 'assembly' is used in the Solidity code. No SWC-127

No potential SWC-100 vulnerability gas limit issues found.

No potential SWC-129 vulnerability found.

False SWC-130

SWC-131 vulnerability discovered. Unused variables found:

- fee

No SWC-132

No SWC-133

False SWC-134

No effect-free code found. No SWC-135

SWC-136 vulnerability discovered. Unencrypted private data found:

- 0
- 0
- 10
- 300
- 0

Висновки за розділом 3

У цьому розділі ми виконали глибокий аналіз важливих аспектів, пов'язаних з розробкою сканера смарт-контрактів. Почали ми з порівняльного аналізу мов програмування, які можна використовувати для написання сканера, включаючи C, C++ та Python3. Цей порівняльний аналіз дозволив нам з'ясувати переваги та обмеження кожної з цих мов у контексті створення ефективного та надійного сканера.

Далі ми звернули увагу на мови програмування, призначені спеціально для розробки смарт-контрактів, такі як Solidity, Vyper та Yul. Проведений аналіз допоміг визначити їхні особливості та відмінності, що дозволить нам вибрати найбільш підходящий інструмент для наших потреб.

У наступній частині ми детально описали теоретично вразливості SWC-100 - SWC-136, щоб краще розуміти потенційні ризики та небезпеки, які можуть виникнути в смарт-контрактах.

Завершили ми цей розділ, реалізуючи власний сканер смарт-контрактів і розробивши відповідний код для пошуку вразливостей SWC-100 - SWC-136. Це

значно підвищить безпеку та надійність наших смарт-контрактів, дозволяючи вчасно виявляти та усувати потенційні проблеми.

ВИСНОВКИ

Поглиблене вивчення смарт-контрактів розкриває багате полотно технологічних інновацій, регуляторної динаміки та трансформаційного потенціалу. Смарт-контракти, як показав ретельний аналіз визначень, є цифровими угодами, які є автономними, самодостатніми та можуть бути виконані в рамках децентралізованих мереж блокчейн. Таке фундаментальне розуміння відкриває шлях до всебічного вивчення їхнього багатогранного впливу на різні індустрії.

Ефіріум, який позиціонується як передова блокчейн-платформа, слугує стрижнем, що каталізує широке впровадження смарт-контрактів. Його адаптивна архітектура проникла в різні сектори, відкриваючи можливості для підвищення ефективності та прищеплюючи прозорість і довіру до традиційно непрозорих процесів. Реальні приклади підкреслюють відчутні переваги смарт-контрактів Ethereum - від спрощених фінансових транзакцій до покращеного управління ланцюжками поставок та незмінних медичних записів.

Історичний шлях смарт-контрактів Ethereum, простежений від їх зародження до сучасних застосувань, дає критично важливе розуміння їх еволюції та зрілості. Більше того, вивчення нових тенденцій та інновацій, включаючи інтеграцію DeFi та нові випадки використання в іграх та управлінні ідентифікацією, дозволяє спрогнозувати майбутнє, в якому смарт-контракти пронизуватимуть усі аспекти цифрової взаємодії.

Водночас, пошук універсального стандарту безпеки смарт-контрактів відкриває ландшафт, що характеризується різноманітністю та складністю. Хоча єдине рішення залишається недосяжним, безліч найкращих практик, експертних висновків та відповідей регуляторів пропонують дорожню карту як для розробників, так і для юристів-практиків. Глобальне регуляторне середовище, прикладом якого є проактивні ініціативи в таких юрисдикціях, як Великобританія та Австралія, підкреслює необхідність адаптивного управління в тандемі з технологічним прогресом.

У перспективі конвергенція технологічних інновацій та регуляторного управління визначатиме траєкторію розвитку смарт-контрактів. Постійна адаптація та інновації мають першорядне значення для забезпечення цілісності та безпеки смарт-контрактів, тим самим сприяючи зміцненню довіри до зростаючої екосистеми блокчейну. Заохочуючи співпрацю, пильність і прихильність передовим практикам, зацікавлені сторони можуть орієнтуватися в мінливому ландшафті і використовувати весь потенціал смарт-контрактів, щоб переосмислити традиційні бізнес-парадигми і відкрити нову еру децентралізованої співпраці та обміну без довіри.

Наше комплексне дослідження охоплює складну сферу розробки платформи Ethereum та безпеки смарт-контрактів з найдрібнішими деталями. Почавши з вичерпного вивчення базових компонентів Ethereum, ми проаналізували віртуальну машину Ethereum (EVM) та мову програмування Solidity, які є ключовими для створення надійних смарт-контрактів. З огляду на це, ми підкреслили необхідність оволодіння цими елементами для створення ефективних та стійких блокчейн-додатків.

Заглиблюючись далі, ми детально розглянули нюанси взаємодії з сервісами Ethereum, прояснивши такі важливі поняття, як оптимізація використання газу та модернізація транзакцій. Надаючи читачам практичні знання та реальні стратегії, ми допомагаємо їм впевнено орієнтуватися в складнощах розвитку Ethereum. Крім того, наше обговорення оновлень Ethereum гарантує, що читачі залишатимуться в курсі еволюції платформи, сприяючи розвитку культури безперервного навчання та адаптації.

Водночас, наше дослідження розробки сканерів смарт-контрактів розгортається в кропіткій подорожі. Ми почали з порівняльного аналізу мов програмування, аналізуючи їхні сильні та слабкі сторони в контексті створення сканерів. Перейшовши до мов, що спеціалізуються на смарт-контрактах, а саме Solidity, Vyper та Yul, ми розглянули їхні унікальні особливості, щоб допомогти читачам прийняти обґрунтоване рішення відповідно до вимог їхніх проектів.

Далі ми заглибилися в теоретичні вразливості, закладені в SWC-100 до SWC-136, висвітливши потенційні ризики, що ховаються в смарт-контрактах. Озброївшись

цими знаннями, ми приступили до розробки спеціального сканера смарт-контрактів, ретельно розробляючи код для виявлення вразливостей і зміцнення безпеки смарт-контрактів. Такий практичний підхід не лише поглиблює розуміння, але й дає можливість фахівцям проактивно зменшувати ризики та зміцнювати цілісність своїх блокчейн-додатків.

Таким чином, наш вичерпний аналіз і практичні рекомендації складають всеосяжний арсенал як для розробників, так і для аудиторів. Поєднуючи теоретичні знання з практичним застосуванням, ми надаємо зацікавленим сторонам інструменти та знання, необхідні для того, щоб з легкістю та впевненістю орієнтуватися в тонкощах розробки Ethereum та безпеки смарт-контрактів. Завдяки безперервному навчанню, адаптації та пильності ми прокладаємо шлях до майбутнього, в якому блокчейн-додатки процвітатимуть на основі безпеки, надійності та довіри.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Book-Smart, Not Street-Smart: Blockchain-Based Smart Contracts and The Social Workings of Law KAREN E. C. LEVY1 [Електронний ресурс] – Режим доступу: https://www.researchgate.net/publication/313843942_Book-Smart_Not_Street-Smart_Blockchain-Based_Smart_Contracts_and_The_Social_Workings_of_Law
2. Solidity — Solidity 0.8.20 documentation [Електронний ресурс] – Режим доступу: https://docs.soliditylang.org/en/v0.8.20/
3. Introduction to Blockchain, Smart Contracts and Decentralized Applications [Електронний ресурс] – Режим доступу: https://www.researchgate.net/publication/370120309_Introduction_to_Blockchain_Smart_Contracts_and_Decentralized_Applications
4. The Economics of Smart Contracts Kirk Baird* The University of Sydney, Seongho Jeong* Yonsei University, Yeonsoo Kim* Yonsei University, Bernd Burgstaller Yonsei University, Bernhard Scholz The University of Sydney Bernhard. [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/1910.11143
5. Матеріали XII Міжнародної науково-практичної конференції молодих учених та студентів «АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ» – Тернопіль, 6-7 грудня 2023 року 336 УДК 004.43 А. В. Семак, С.-З. Ю. Хома, к.т.н. Г. В. Козбур (Тернопільський національний технічний університет імені Івана Пулюя, Україна) ВИКОРИСТАННЯ СМАРТ-КОНТРАКТІВ ДЛЯ ОПТИМІЗАЦІЇ ПРОЦЕСУ ГОЛОСУВАННЯ НА ВИБОРАХ [Електронний ресурс] – Режим доступу: https://elartu.tntu.edu.ua/browse?type=author&value=%D0%9A%D0%BE%D0%B7%D0%B1%D1%83%D1%80%2C+%D0%93%D0%B0%D0%BB%D0%B8%D0%BD%D0%B0+%D0%92%D0%BE%D0%BB%D0%BE%D0%B4%D0%B8%D0%BC%D0%B8%D1%80%D1%96%D0%B2%D0%BD%D0%B0

6. What Do We Mean by Smart Contracts? Open Challenges in Smart Contracts Maria G. Vigliotti [Електронний ресурс] – Режим доступу: https://www.frontiersin.org/articles/10.3389/fbloc.2020.553671/full
7. N. Szabo. (1994). Smart Contracts. [Online]. [Електронний ресурс] – Режим доступу: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literat%
8. Recent Advances in Smart Contracts: A Technical Overview and State of the Art VICTOR YOUDOM KEMMOE 1 , WILLIAM STONE 1 , JEEHYEONG KIM 2, DAEYOUNG KIM 1 , AND JUNGGAB SON 1 , (Member, IEEE) [Електронний ресурс] – Режим доступу: https://www.researchgate.net/publication/342456186_Recent_Advances_in_Smart_Contracts_A_Technical_Overview_and_State_of_the_Art
9. ETHEREUM 2.0: HOW IT WORKS AND WHY IT MATTERS Christine Kim [Електронний ресурс] – Режим доступу: https://cognizium.io/uploads/resources/file-844047896485.pdf
10. DEVELOPER REPORT [Електронний ресурс] – Режим доступу: https://www.developerreport.com/
11. Законопроект № 7183 зареєстрований у парламенті 6 жовтня. [Електронний ресурс] – Режим доступу: https://ukrinform.ua/rubric-economy/2319825-radi-proponuut-prijnati-zakon-pro-kriptovaluti.html
12. Payment Services Act [Електронний ресурс] – Режим доступу: https://www.japaneselawtranslation.go.jp/en/laws/view/3078/en
13. Smart contracts [Електронний ресурс] – Режим доступу: https://lawcom.gov.uk/project/smart-contracts/
14. Розуміння віртуальної машини ethereum evm [Електронний ресурс] – Режим доступу: https://www.securities.io/uk/%D1%80%D0%BE%D0%B7%D1%83%D0%BC%D1%96%D0%BD%D0%BD%D1%8F-%D0%B2%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D

0%BD%D0%BE%D1%97-
%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B8-ethereum-evm/

15. SMART CONTRACT SECURITY [Електронний ресурс] – Режим доступу: https://ethereum.org/en/developers/docs/smart-contracts/security/
16. Contracts 4.x [Електронний ресурс] – Режим доступу: https://docs.openzeppelin.com/contracts/4.x/
17. Contracts 2.x [Електронний ресурс] – Режим доступу: https://docs.openzeppelin.com/contracts/2.x/api/math
18. How to Get the Balance of an ERC-20 Token Using Web3.js [Електронний ресурс] – Режим доступу: https://www.quicknode.com/guides/ethereum-development/smart-contracts/how-to-get-the-balance-of-an-erc-20-token
19. How to Create and Deploy an Upgradeable ERC20 Token [Електронний ресурс] – Режим доступу: https://www.quicknode.com/guides/ethereum-development/smart-contracts/how-to-create-and-deploy-an-upgradeable-erc20-token
20. Detect MetaMask [Електронний ресурс] – Режим доступу: https://docs.metamask.io/wallet/how-to/connect/detect-metamask/
21. More What is Uniswap and 0x? [Електронний ресурс] – Режим доступу: https://walletopinion.com/101/uniswap-0x-dex/
22. Connecting the world to blockchains [Електронний ресурс] – Режим доступу: https://chain.link/
23. What is Mythril? [Електронний ресурс] – Режим доступу: https://mythril-classic.readthedocs.io/en/master/about.html
24. What is Slither? [Електронний ресурс] – Режим доступу: https://www.alchemy.com/dapps/slither
25. What is Oyente? [Електронний ресурс] – Режим доступу: https://www.alchemy.com/dapps/oyente
26. SmartCheck [Електронний ресурс] – Режим доступу: https://github.com/smartdec/smartcheck
27. Ethereum Smart Contract Security Best Practices [Електронний ресурс] – Режим доступу: https://consensys.github.io/smart-contract-best-practices/

28. Benefits of a Smart Contract Audit and Diligence's Ethereum Security Service [Електронний ресурс] – Режим доступу: https://consensys.io/diligence/
29. Stack Exchange [Електронний ресурс] – Режим доступу: https://ethereum.stackexchange.com/
30. Journal of Information & Communication Technology - JICT Vol. 14 Issue. 1
31. Solidity — Solidity SWC-100 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-100/
32. Solidity — Solidity SWC-101 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-101/
33. Solidity — Solidity SWC-102 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-102/
34. Solidity — Solidity SWC-103 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-103/
35. Solidity — Solidity SWC-104 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-104/
36. Solidity — Solidity SWC-105 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-105/
37. Solidity — Solidity SWC-106 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-106/
38. Solidity — Solidity SWC-107 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-107/
39. Solidity — Solidity SWC-108 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-108/
40. Solidity — Solidity SWC-109 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-109/
41. Solidity — Solidity SWC-110 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-110/
42. Solidity — Solidity SWC-111 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-111/

43. Solidity — Solidity SWC-112 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-112/
44. Solidity — Solidity SWC-113 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-113/
45. Solidity — Solidity SWC-114 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-114/
46. Solidity — Solidity SWC-115 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-115/
47. Solidity — Solidity SWC-116 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-116/
48. Solidity — Solidity SWC-117 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-117/
49. Solidity — Solidity SWC-118 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-118/
50. Solidity — Solidity SWC-119 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-119/
51. Solidity — Solidity SWC-120 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-120/
52. Solidity — Solidity SWC-121 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-121/
53. Solidity — Solidity SWC-122 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-122/
54. Solidity — Solidity SWC-123 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-123/
55. Solidity — Solidity SWC-124 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-124/
56. Solidity — Solidity SWC-125 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-125/
57. Solidity — Solidity SWC-126 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-126/

58. Solidity — Solidity SWC-127 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-127/
59. Solidity — Solidity SWC-128 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-128/
60. Solidity — Solidity SWC-129 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-129/
61. Solidity — Solidity SWC-130 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-130/
62. Solidity — Solidity SWC-131 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-131/
63. Solidity — Solidity SWC-132 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-132/
64. Solidity — Solidity SWC-133 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-133/
65. Solidity — Solidity SWC-134 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-134/
66. Solidity — Solidity SWC-135 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-135/
67. Solidity — Solidity SWC-136 [Електронний ресурс] – Режим доступу: https://swcregistry.io/docs/SWC-136/

ДОДАТОК А

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Статті у наукових фахових виданнях України

1. VII Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)” Maryana Levitska, Yaroslav Kulaha, Serhiy Tolyupa DEVELOPMENT OF RECOMMENDATIONS FOR THE SECURITY OF SMART CONTRACTS
2. IT&I 2023 Information Technology and Interactions 21 листопада 2023 «VULNERABILITY SCANNER FOR BLOCKCHAIN SMART CONTRACTS» Serhii Toliupa Yaroslav Kulaha Valeriia Solodovnyk

ДОДАТОК Б

Лістинг коду програми

```
import re
def swc136(solidity_code):
    pattern = r"(private\s+)(.*?)\s*=\s*(.*?)\s*;"
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Unencrypted private data found:")
        for match in matches:
            print(f"- {match[2]}")
        print("\nIt's important that unencrypted private data is not stored in the contract code or state.")
    else:
        print("No unencrypted private data found.")
def swc135(solidity_code):
    pattern = r"msg\.sender\.call\.value\(.*\)\s*\(\)\s*"
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Effect-free code found:")
        for match in matches:
            print(f"- {match}")
        print("\nIt's recommended to check the return value of the call to ensure the intended effect is being produced.")
    else:
        print("No effect-free code found.")
def swc134(solidity_code):
    pattern = r"(\._callable\.(transfer|send|call\.value|call))\((.*?)\)"
    matches = re.findall(pattern, solidity_code)
```

```
    if matches:
        print("The transfer() and send() functions forward a fixed amount of 2300 gas.")
        print("Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks.")
        print("However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs.")
        print("For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.")
        return True
    return  print("False swc134")
def swc133(solidity_code):
    pattern = r"abi\.encodePacked\((.*?)\)"
    matches = re.findall(pattern, solidity_code)
    for match in matches:
        if len(match.strip().split(",")) > 1 and any(isinstance(i, list) for i in match.strip().split(",")):
            print("Using abi.encodePacked() with multiple variable length arguments can lead to a hash collision.")
            return True
    return  print("False swc133")
def swc132(solidity_code):
    if "assert(address(lockAddr).balance == msg.value)" in solidity_code:
        print("The code assumes a specific Ether balance. This can lead to DOS conditions.")
        return True
    return print("False swc132")
def swc131(solidity_code):
    unused_variables = []
    lines = solidity_code.split("\n")
    variables = set()
```

```
    for line in lines:
        match = re.search(r"(\w+)(\s+)(\w+)", line)
        if match:
            variables.add(match.group(1))
    for line in lines:
        if "=" not in line:
            continue
        match = re.search(r"(\w+)\s*=", line)
        if match:
            variable = match.group(1)
            if variable in variables and variable not in unused_variables:
                unused_variables.append(variable)
    return unused_variables
def swc130(solidity_code):
    rtl_override_unicode = "\u202E"
    if rtl_override_unicode in solidity_code:
        print(f"RTL override unicode character `{rtl_override_unicode}` found in the code. Please remove it to prevent confusion.")
        return True
    return print("False swc130")
def swc129(solidity_code):
    typos = []
    pattern = r"(\w+)\s*=\+"
    matches = re.findall(pattern, solidity_code)
    for match in matches:
        typos.append(f"Potential typo found: `{match} =+` instead of `+=`")
    return typos
def swc128(solidity_code):
    issues = []
    pattern = r"(\w+)\[\]\.push\(.*\)"
```

```
    matches = re.findall(pattern, solidity_code)
    for match in matches:
        issues.append(f"Potential gas limit issue found: `{match}[]` may grow indefinitely and cause a Denial of Service condition.")
    return issues
def swc127(solidity_code):
    assembly_pattern = r"assembly\s+{"
    assembly_use = re.findall(assembly_pattern, solidity_code)
    if assembly_use:
        print("Solidity supports function types. That is, a variable of function type can be assigned with a reference to a function with a matching signature.")
        print("The problem arises when a user has the ability to arbitrarily change the function type variable and thus execute random code instructions.")
        print("As Solidity doesn't support pointer arithmetics, it's impossible to change such variable to an arbitrary value.")
        print("However, if the developer uses assembly instructions, such as mstore or assign operator, in the worst case scenario an attacker is able to point a function type variable to any code instruction, violating required validations and required state changes.")
        print("The following contracts use 'assembly' in the Solidity code:")
        print(", ".join(assembly_use))
    else:
        print("No 'assembly' is used in the Solidity code.")
def swc126(solidity_code):
    call_pattern = r"address\..*\.call\("
    call_use = re.findall(call_pattern, solidity_code)
    if call_use:
        print("Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract.")
        print("If the sub-call fails, either the whole transaction is reverted, or execution is continued.")
```

```
        print("In the case of a relayer contract, the user who executes the transaction, the 'forwarder', can effectively censor transactions by using just enough gas to execute the transaction, but not enough for the sub-call to succeed.")
        print("The following contracts use 'call' or 'delegatecall' in the Solidity code:")
        print(", ".join(call_use))
    else:
        print("No 'call' or 'delegatecall' is used in the Solidity code.")
def swc125(solidity_code):
    multiple_inheritance_pattern = r"contract\s+[A-Za-z0-9_]+\s+is\s+[A-Za-z0-9_\s,]+\s+"
    multiple_inheritance_use = re.findall(multiple_inheritance_pattern, solidity_code)
    if multiple_inheritance_use:
     print("Solidity supports multiple inheritance, meaning that one contract can inherit several contracts.")
        print("Multiple inheritance introduces ambiguity called Diamond Problem: if two or more base contracts define the same function, which one should be called in the child contract?")
     print("Solidity deals with this ambiguity by using reverse C3 Linearization, which sets a priority between base contracts.")
        print("That way, base contracts have different priorities, so the order of inheritance matters.")
    print("Neglecting inheritance order can lead to unexpected behavior.")
        print("The following contracts use multiple inheritance in the Solidity code:")
        print(", ".join(multiple_inheritance_use))
    else:
        print("No multiple inheritance is used in the Solidity code.")
def swc124(solidity_code):
    storage_pattern = r"(?:mapping|address|uint|int|bool|bytes|string|struct|enum)\s+(\w+)(?=\s*[\[\(])"
    storage_use = re.findall(storage_pattern, solidity_code)
    if storage_use:
```

```
        print("A smart contract's data is persistently stored at some storage location on the EVM level.")
        print("The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations.")
        print("If an attacker is able to write to arbitrary storage locations of a contract, the authorization checks may easily be circumvented.")
        print("This can allow an attacker to corrupt the storage; for instance, by overwriting a field that stores the address of the contract owner.")
        print("The following storage locations are used in the Solidity code:")
        print(", ".join(storage_use))
    else:
        print("No storage locations are used in the Solidity code.")
def swc123(solidity_code):
    pattern = r"require\s*\((.*?)\)\s*;"
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Potential issues with require() construct found:")
        for match in matches:
            print(f"- {match}")
        print("\nThe Solidity require() construct is meant to validate external inputs of a function. In most cases, such external inputs are provided by callers, but they may also be returned by callees. Violations of a requirement can indicate one of two possible issues:")
        print("- A bug exists in the contract that provided the external input.")
        print("- The condition used to express the requirement is too strong.")
    else:
        print("No potential issues with require() construct found.")
def swc122(solidity_code):
    pattern = r"(msg\.sender\s*==\s*(?:ecrecover\(.*?\)\s*|\s*address\(\s*sha3\(.*?\)\s*\)))"
    matches = re.findall(pattern, solidity_code)
    if matches:
```

```
    print("Potential issues with signature verification found:")
    for match in matches:
        print(f"- {match}")
    print("\nIt is a common pattern for smart contract systems to allow users to sign messages off-chain instead of directly requesting users to do an on-chain transaction because of the flexibility and increased transferability that this provides. Smart contract systems that process signed messages have to implement their own logic to recover the authenticity from the signed messages before they process them further. A limitation for such systems is that smart contracts can not directly interact with them because they can not sign messages. Some signature verification implementations attempt to solve this problem by assuming the validity of a signed message based on other methods that do not have this limitation. An example of such a method is to rely on msg.sender and assume that if a signed message originated from the sender address then it has also been created by the sender address. This can lead to vulnerabilities especially in scenarios where proxies can be used to relay transactions.")
  else:
    print("No potential issues with signature verification found.SWC122")
def swc121(solidity_code):
  pattern = r"(ecrecover\(.*?\)\s*==\s*(?:msg\.sender\s*|\s*address\(\s*sha3\(.*?\)\s*\)))"
  matches = re.findall(pattern, solidity_code)
  if matches:
    print("Potential issues with signature verification found:")
    for match in matches:
      print(f"- {match}")
    print("\nIt is sometimes necessary to perform signature verification in smart contracts to achieve better usability or to save gas cost. A secure implementation needs to protect against Signature Replay Attacks by for example keeping track of all processed message hashes and only allowing new message hashes to be processed. A malicious user could attack a contract without such a control and get message hash that was sent by another user processed multiple times.")
```

```
    else:
        print("No potential issues with signature verification found.")
def swc120(solidity_code):
    pattern = r"(block\.blockhash\(.*?\))"
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Potential issues with random number generation found:")
        for match in matches:
            print(f"- {match}")
        print("\nAbility to generate random numbers is very helpful in all kinds of applications. One obvious example is gambling DApps, where pseudo-random number generator is used to pick the winner. However, creating a strong enough source of randomness in Ethereum is very challenging. For example, use of block.timestamp is insecure, as a miner can choose to provide any timestamp within a few seconds and still get his block accepted by others. Use of blockhash, block.difficulty and other fields is also insecure, as they're controlled by the miner. If the stakes are high, the miner can mine lots of blocks in a short time by renting hardware, pick the block that has required block hash for him to win, and drop all others.")
    else:
        print("No potential issues with random number generation found.")
def swc119(solidity_code):
    patterns = [
        r'\+(?=([^=]|$))',
        r'-(?=([^=]|$))',
        r'\*(?=([^=]|$))',
        r'\/(?=([^=]|$))',
        r'%(?=([^=]|$))',
    ]
    warnings = {}
     for line_num, line in enumerate(solidity_code.split('\n')):
        for pattern in patterns:
```

```
        matches = re.finditer(pattern, line)
        for match in matches:
            op = match.group()
            surrounding_code = line[max(0, match.start() - 10):match.end() + 10]
            if re.search(r'(?<=\s|^)(return|:=|=\s*)', surrounding_code):
                warnings[f"Line {line_num + 1}"] = f"Potential overflow/underflow in {op} operation: {surrounding_code}"
    return warnings
def swc118(solidity_code):
    constructor_pattern = r"constructor\s*\((?![^(]*\btransaction_malleablity\b)[^)]*\)\s*{"
    constructor_use = re.search(constructor_pattern, solidity_code)
    if constructor_use:
        print("A constructor with a name different from the contract name is used in the Solidity code.")
        print("Before Solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract class containing it.")
        print("A function meant to become a constructor becomes a normal, callable function if its name doesn't exactly match the contract name.")
        print("This behavior sometimes leads to security issues, inparticular when smart contract code is re-used with a different name but the name of the constructor function is not changed accordingly.")
    else:
        print("A constructor with a name different from the contract name is not used in the Solidity code.")
def swc117(solidity_code):
    ecrecover_pattern = r"(?<=\b)ecrecover(?=\b)"
    ecrecover_use = re.search(ecrecover_pattern, solidity_code)
    if ecrecover_use:
        print("The 'ecrecover' function is used in the Solidity code.")
        print("The implementation of a cryptographic signature system in Ethereum contracts
```

```
often assumes that the signature is unique, but signatures can be altered without the possession of the private key and still be valid.")
        print("A malicious user can slightly modify the three values v, r and s to create other valid signatures.")
        print("A system that performs signature verification on contract level might be susceptible to attacks if the signature is part of the signed message hash.")
        print("Valid signatures could be created by a malicious user to replay previously signed messages.")
    else:
        print("The 'ecrecover' function is not used in the Solidity code.")
def swc116(solidity_code):
    time_value_pattern = r"(?<=\b)(block\.(timestamp|number))(?=\b)"
    time_value_use = re.findall(time_value_pattern, solidity_code)
    if time_value_use:
        print("The following time values are used in the Solidity code:")
        for time_value in time_value_use:
            print(f"- {time_value}")
            if time_value == "block.timestamp":
                print("  - Using 'block.timestamp' for time-dependent events could make a contract vulnerable.")
                print("  - The timestamp can be altered by malicious miners, especially if they can gain advantages by doing so.")
                print("  - Developers can't rely on the preciseness of the provided timestamp.")
            if time_value == "block.number":
                print("  - Using 'block.number' for precise calculations of time could make a contract vulnerable.")
                print("  - Block times are not constant and are subject to change for a variety of reasons.")
    else:
        print("The 'block.timestamp' and 'block.number' variables are not used in the Solidity
```

```
code.")
def swc115(solidity_code):
    tx_origin_pattern = r"(?<=\b)tx\.origin(?=\b)"
    tx_origin_use = re.search(tx_origin_pattern, solidity_code)
    if tx_origin_use:
        print("The 'tx.origin' variable is used in the Solidity code.")
        print("Using 'tx.origin' for authorization could make a contract vulnerable.")
        print("A malicious contract could call into the vulnerable contract and bypass the authorization check.")
    else:
        print("The 'tx.origin' variable is not used in the Solidity code.")
def swc114(solidity_code):
    arithmetic_operation_pattern = r"(?<=\b)(add|sub|mul|div|mod|exp|signextend|shiftleft|shiftright)(?=\()"
    arithmetic_operations = re.findall(arithmetic_operation_pattern, solidity_code)
    if arithmetic_operations:
        print("The following arithmetic operations are used:")
        for arithmetic_operation in arithmetic_operations:
            print(f"- {arithmetic_operation}")
            if arithmetic_operation == "sub":
                print("  - Potential race condition in the 'approve' function:")
                print("    - The 'balances[msg.sender] -= value' operation may be vulnerable to a race condition.")
                print("    - A malicious actor may submit a 'transferFrom' request with a higher gas price, causing the transfer to occur before the 'approve' function.")
                print("    - This may result in the malicious actor receiving more tokens than they should have.")
            if arithmetic_operation == "sub":
                print("  - Potential race condition in the 'transferFrom' function:")
                print("    - The 'balances[from] -= value' operation may be vulnerable to a race
```

```
condition.")
            print("    - A malicious actor may submit a 'transferFrom' request with a higher gas price, causing the transfer to occur before the original 'transferFrom' request.")
            print("    - This may result in the malicious actor receiving more tokens than they should have.")
    else:
        print("No arithmetic operations are used in the Solidity code.")
def swc113(solidity_code):
    external_call_pattern = r"(?<=\b)(send|transfer|call\.value|delegatecall|staticcall|callcode|create|create2)(?=\()"
    external_calls = re.findall(external_call_pattern, solidity_code)
    if external_calls:
        print("The following external calls are used:")
        for external_call in external_calls:
            print(f"- {external_call}")
    else:
        print("No external calls are used in the Solidity code.")
def swc112(solidity_code):
    delegatecall_pattern = r"(?<=\bdelegatecall\()(.*?)(?=\))"
    delegatecalls = re.findall(delegatecall_pattern, solidity_code)
    if delegatecalls:
        print("The delegatecall() function is used in the following locations:")
        for delegatecall in delegatecalls:
            print(f"- {delegatecall}")
    else:
        print("The delegatecall() function is not used in the Solidity code.")
def swc111(solidity_code):
    deprecated_pattern = r"(?<=\b)(blockhash|keccak256|gasleft|delegatecall|selfdestruct)(?=\b)"
    deprecated_calls = re.findall(deprecated_pattern, solidity_code)
```

```
    if deprecated_calls:
        print("The following deprecated functions and operators are used:")
        for call in deprecated_calls:
            print(f"- {call}")
    else:
        print("No deprecated functions or operators are used in the Solidity code.")
def swc110(solidity_code):
    assert_pattern = r"(?<=\bassert\()(.*?)(?=\);)"
    assert_calls = re.findall(assert_pattern, solidity_code)
    if assert_calls:
        print("The assert() function is used in the following locations:")
        for call in assert_calls:
            print(f"- {call}")
    else:
        print("The assert() function is not used in the Solidity code.")
def swc109(solidity_code):
    pattern = r'(?:uint|address|bytes|bool|string|mapping|struct|enum)\s+([a-zA-Z_][a-zA-Z_0-9]*)'
    warnings = {}
    for line_num, line in enumerate(solidity_code.split('\n')):
        matches = re.finditer(pattern, line)
        for match in matches:
            var_name = match.group(1)
            if not re.search(rf'\b{var_name}\s*=', line):
                warnings[f"Line {line_num + 1}"] = f"Uninitialized local storage variable: {var_name}"
    return warnings
def swc108(solidity_code):
    pattern = r'(?:uint|address|bytes|bool|string|mapping|struct|enum)\s+([a-zA-Z_][a-zA-Z_0-9]*)'
```

```
    warnings = {}
    for line_num, line in enumerate(solidity_code.split('\n')):
        matches = re.finditer(pattern, line)
        for match in matches:
            var_name = match.group(1)
            if not re.search(r'\b(public|private|internal)\s+', line):
                warnings[f"Line {line_num + 1}"] = f"Variable {var_name} does not have explicit visibility."
    return warnings
def swc107(solidity_code):
    pattern = r'function\s+(\w+)\s+(\w+)\s*\((.*)\)\s*(external)?\s*(public|private|internal|external)?\s*(override)?\s*(pure|view)?'
    warnings = {}
    for line_num, line in enumerate(solidity_code.split('\n')):
        matches = re.finditer(pattern, line)
        for match in matches:
            func_name = match.group(1)
            func_visibility = match.group(4)
            if match.group(4) == 'external' and match.group(7) not in ['view', 'pure']:
                if re.search(r'(\w+)\.(\w+)\(', line):
                    warnings[f"Line {line_num + 1}"] = f"Potential reentrancy vulnerability: External function {func_name} is not marked as view or pure and calls another contract."
    return warnings
def swc106(solidity_code):
    pattern = r"""selfdestruct\s*\(.*?\);"""
    pattern = re.compile(pattern, re.VERBOSE)
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Potential self-destruct vulnerabilities found:")
```

```
        for match in matches:
            print(f"- {match}")
        print("\nMissing or insufficient access controls can allow malicious parties to self-destruct the contract. To prevent this, it's a good idea to restrict the use of selfdestruct to a privileged address or to a function that can only be called by the contract owner.")
    else:
        print("No potential self-destruct vulnerabilities found.")
def swc105(solidity_code):
    pattern = r"""
    function\s+.*?\(.*?\)\s+payable\s+.*?\s*{
        (?!.*?\brequire\s*\(.*?\bmsg\.sender\s*!=\s*.*?\b\).*?\})
        .*?
    }
    """
    pattern = re.compile(pattern, re.VERBOSE)
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Potential Ether withdrawal vulnerabilities found:")
        for match in matches:
            print(f"- {match}")
        print("\nMissing or insufficient access controls can allow malicious parties to withdraw some or all Ether from the contract account. This bug is sometimes caused by unintentionally exposing initialization functions. By wrongly naming a function intended to be a constructor, the constructor code ends up in the runtime byte code and can be called by anyone to re-initialize the contract. To prevent this, it's a good idea to restrict the use of payable functions to privileged addresses or to a function that can only be called by the contract owner.")
    else:
        print("No potential Ether withdrawal vulnerabilities found.")
def swc104(solidity_code):
```

```
    pattern = r"""
    function\s+.*?\(address\s+callee\).*?\s*{
        (?!.*?\brequire\s*\(.*?\bcallee\.call\(.*?\)\s*==\s*true\b.*?\})
        .*?
    }
    """
    pattern = re.compile(pattern, re.VERBOSE)
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("Potential unchecked message call vulnerabilities found:")
        for match in matches:
            print(f"- {match}")
        print("\nThe return value of a message call is not checked. Execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behaviour in the subsequent program logic. To prevent this, it's a good idea to check the return value of message calls and ensure that they are successful before continuing execution.")
    else:
        print("No potential unchecked message call vulnerabilities found.")
def swc103(solidity_code):
    pattern = r"""
    pragma\s+solidity\s+[><=!~*]*\s*[0-9]*\.*[0-9]*\.*[0-9]*[><=!~*]*\s*;
    """
    pattern = re.compile(pattern, re.VERBOSE)
    matches = re.findall(pattern, solidity_code)
    if matches:
        print("No potential pragma version vulnerabilities found.")
    else:
        print("Potential pragma version vulnerabilities found:")
        for match in matches:
```

```
            print(f"- {match}")
        print("\nContracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. To prevent this, it's a good idea to lock the pragma to a specific version or range of versions.")
def swc102(solidity_code):
    pattern = r"""pragma\s+solidity\s+[0-9]*\.*[0-9]*\.*[0-9]*\s*;"""
    pattern = re.compile(pattern, re.VERBOSE)
    match = re.search(pattern, solidity_code)
    if match:
        pragma_version = match.group(0)[10:].strip()
        if pragma_version < "0.8.0":
            print(f"Potential outdated compiler version vulnerability found: {pragma_version}")
            print("\nUsing an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version. To prevent this, it's a good idea to use the latest stable compiler version or a version that has been thoroughly tested and reviewed.")
        else:
            print("No potential outdated compiler version vulnerabilities found.")
    else:
        print("No pragma version found in the Solidity code.")
def swc101(solidity_code):
    pattern = r'(\w+)\s*(\+|-|\*|/|%)\s*(\w+)'
    warnings = {}
    for line_num, line in enumerate(solidity_code.split('\n')):
        matches = re.finditer(pattern, line)
        for match in matches:
            var1 = match.group(1)
```

```
        op = match.group(2)
        var2 = match.group(3)
        if re.search(r'uint\d+|int\d+', var1) or re.search(r'uint\d+|int\d+', var2):
                warnings[f"Line {line_num + 1}"] = f"Potential arithmetic overflow or underflow: Variables {var1} and {var2} are of unsigned or signed integer type and are used in an arithmetic operation with operator {op}."
    return warnings
def swc100(solidity_code):
    pattern = r"""function\s+[a-zA-Z0-9_]+\s*\("""
    pattern = re.compile(pattern, re.VERBOSE)
   matches = re.findall(pattern, solidity_code)
    if matches:
        print("Potential function visibility vulnerabilities found:")
        for match in matches:
            print(f"- {match[:-1]}")
        print("\nFunctions that do not have a function visibility type specified are public by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes. To prevent this, it's a good idea to make a conscious decision on which visibility type is appropriate for a function. This can dramatically reduce the attack surface of a contract system.")
    else:
        print("No potential function visibility vulnerabilities found.")
solidity_code = """
pragma solidity 0.6.4;
interface ICallable {
    function callMe() external;
}
contract HardcodedNotGood {
    address payable _callable = 0xaAaAaAaaAaAaAaaAaAAAAAAAAAaaaAaAaAaaAaaAa;
```

```
    ICallable callable = ICallable(_callable);
    constructor() public payable {
    }
    function doTransfer(uint256 amount) public {
        _callable.transfer(amount);
    }
    function doSend(uint256 amount) public {
        _callable.send(amount);
    }
     function callLowLevel() public {
         _callable.call.value(0).gas(10000)("");
     }
     function callWithArgs() public {
         callable.callMe{gas: 10000}();
     }
}
"""
swc100(solidity_code)
warnings = swc101(solidity_code)
if warnings:
    print("Arithmetic overflow/underflow warnings:")
    for line, warning in warnings.items():
        print(f"{line}: {warning}")
else:
    print("No arithmetic overflow/underflow warnings found.")
swc102(solidity_code)
swc103(solidity_code)
swc104(solidity_code)
swc105(solidity_code)
swc106(solidity_code)
```

```
warnings = swc107(solidity_code)
if warnings:
    print("Reentrancy warnings:")
    for line, warning in warnings.items():
        print(f"{line}: {warning}")
else:
    print("No reentrancy warnings found.")
warnings = swc108(solidity_code)
if warnings:
    print("Variable visibility warnings:")
    for line, warning in warnings.items():
        print(f"{line}: {warning}")
else:
    print("No variable visibility warnings found.")
warnings = swc109(solidity_code)
if warnings:
    print("Uninitialized local storage variable warnings:")
    for line, warning in warnings.items():
        print(f"{line}: {warning}")
else:
    print("No uninitialized local storage variable warnings found.")
swc110(solidity_code)
swc111(solidity_code)
swc112(solidity_code)
swc113(solidity_code)
swc114(solidity_code)
swc115(solidity_code)
swc116(solidity_code)
swc117(solidity_code)
swc118(solidity_code)
```

```
warnings = swc119(solidity_code)
if warnings:
    print("Potential overflow/underflow warnings:")
    for line, warning in warnings.items():
        print(f"{line}: {warning}")
else:
    print("No potential overflow/underflow warnings found.119")
swc120(solidity_code)
swc121(solidity_code)
swc122(solidity_code)
swc123(solidity_code)
swc124(solidity_code)
swc125(solidity_code)
swc126(solidity_code)
swc127(solidity_code)
issues = swc128(solidity_code)
if issues:
    print("Potential gas limit issues found:")
    for issue in issues:
        print(issue)
else:
    print("No potential gas limit issues found. swc128")
typos = swc129(solidity_code)
if typos:
    print("Potential typos found:")
    for typo in typos:
        print(typo)
else:
    print("No potential typos found. swc-129")
swc130(solidity_code)
```

```
unused_variables = swc131(solidity_code)
if unused_variables:
    print("Unused variables found:")
    for variable in unused_variables:
        print(f"- {variable}")
else:
    print("No unused variables found. swc-131")
swc132(solidity_code)
swc133(solidity_code)
swc134(solidity_code)
swc135(solidity_code)
swc136(solidity_code)
```

ДОДАТОК В

Trilight Security OÜ

EU, Estonia, Tallinn, Harju maakond,
Kesklinnalinnaosa, Vesivärava tn 50-201, 10126
Tel: +372 54550868; email: connect@trilightsecurity.com

АКТ ВПРОВАДЖЕННЯ

Сканер вразливостей SWC-100-136, розроблений Ярославом Кулагою, виявився ефективним інструментом для проведення тестів на проникнення смарт-контрактів у комерційних проектах. Використання цього сканера дозволило нам ефективно виявити потенційні вразливості в смарт-контрактах.

Під час декількох проектів завдяки цьому сканеру було виявлено кілька вразливостей різних рівнів. Наприклад, в одному з проектів була виявлена вразливість SWC-100 високого рівня, коли було з’ясовано, що розробник не вказав тип видимості функції як "private".

Крім того, були виявлені вразливості середнього рівня, такі як SWC-132 і SWC-115, а також вразливості низького рівня, такі як SWC-117, SWC-110, SWC-109 і SWC-133.

Можемо рекомендувати використання сканеру вразливостей SWC-100-136 для вирішення виробничих завдань під час проектів, пов’язаних з оцінюванням рівня кібербезпеки блокчейн-рішень.

Ян Шмиголь
Генеральний директор
Trilight Security OÜ