Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
Факультет інформаційних технологій
Кафедра безпеки інформації та телекомунікацій

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеню магістра

студента *Синявського Сергія Сергійовича*
академічної групи *125м-20-2*
спеціальності *125 Кібербезпека*
спеціалізації[1]
за освітньо-професійною програмою *Кібербезпека*
на тему *Способи розробки кібербезпечних смарт-контрактів у блокчейні Ethereum*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	к.т.н. доц. Сафаров О.О.			
розділів:				
спеціальний	к.т.н. доц. Сафаров О.О.			
економічний	к.е.н., доц. Пілова Д.П.			

Рецензент				

Нормоконтролер	ст. викл. Тимофєєв Д.С.			

Дніпро
2022

ЗАТВЕРДЖЕНО:

завідувач кафедри
безпеки інформації та телекомунікацій
____________ д.т.н., проф. Корнієнко В.І.

«_____»_________________20___року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня магістра

студенту ***Синявському Сергію Сергійовичу*** **академічної групи** ***125м-20-2***
(прізвище ім'я по-батькові) (шифр)

спеціальності *125 Кібербезпека*
(код і назва спеціальності)

на тему ***Способи розробки кібербезпечних смарт-контрактів у блокчейні Ethereum***

затверджену наказом ректора НТУ «Дніпровська політехніка» від 10.12.2021 № 1036-с

Розділ	Зміст	Термін виконання
Розділ 1	Надання основної інформації про блокчейн, смарт-контракти та програмне забезпечення для їх розробки.	01.11.2021
Розділ 2	Приведення інформації про основні вразливості смарт-контрактів, їх пояснення та демонстрація прикладів їх реалізації.	20.11.2021
Розділ 3	Проведення аудиту демонстраційного смарт-контракту, надання рекомендацій щодо протидії виявленим вразливостям та їх програмна реалізація.	15.12.2021
Розділ 4	Розрахунок економічної доцільності реалізації методів захисту смарт-контракту від виявлених вразливостей.	10.01.2022

Завдання видано ____________________ ______________
(підпис керівника) (прізвище, ініціали)

Дата видачі: 01.10.2021

Дата подання до екзаменаційної комісії: 19.01.2022

Прийнято до виконання ____________________ ______________
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 74 с., 31 рис., 4 табл., 9 додатків, 20 джерел.

Об‘єкт дослідження: Смарт-контракти в блокчейні Ethereum.

Предмет дослідження: Кібербезпека смарт-контрактів в блокчейні Ethereum при їх розробці.

Мета кваліфікаційної роботи: Дослідження вразливостей та розробка методів забезпечення захисту смарт-контрактів в блокчейні Ethereum.

В першому розділі була приведена інформація про блокчейн, смарт-коонтракти і програмне забезпечення, необхідне для їх написання та приведений опис демонстраційного смарт-контракту PPSD.sol.

В другому розділі роботи було описано найбільш популярні вразливості смарт-контрактів та приведені приклади їх реалізації.

В спеціальному розділі був проведений аудит смарт-контракту PPSD.sol та реалізація заходів захисту смарт-контракту від виявлених вразливостей.

В четвертому розділі були проведені розрахунки економічної ефективності обраних методів захисту смарт-контракту PPSD.sol.

За результатами даної роботи обраний смарт-контракт набув необхідний функціонал для захисту від вразливостей та став доступний для розгортання в блокчейн. Таким чином наступним кроком буде деплой смарт-контракту в блокчейн Ethereum та виконання його основних функцій.

БЛОКЧЕЙН, СМАРТ-КОНТРАКТ, ВРАЗЛИВІСТЬ, ДЕПЛОЙ, АУДИТ, ТОКЕН, КРИПТОВАЛЮТА, ПРОГРАМУВАННЯ.

ABSTRACT

Explanatory note: 74 p., 31 fig., 4 tab., 9 adds, 20 sources.

Object of study: Smart contracts in the Ethereum blockchain.

Subject of research: Cybersecurity of smart contracts in the Ethereum blockchain during their development.

Objective of the qualification work: Investigation of vulnerabilities and development of methods to ensure the protection of smart contracts in the Ethereum blockchain.

The first section provided information on the blockchain, smart contracts and software needed to write them, and described the PPSD.sol smart contract demonstration contract.

The second section described the most popular vulnerabilities in smart contracts and gave examples of their implementation.

In a special section, the PPSD.sol smart contract was audited and the smart contract was implemented to protect against identified vulnerabilities.

In the fourth section, calculations of the cost-effectiveness of selected methods of protection of the smart contract PPSD.sol were performed.

As a result of this work, the selected smart contract acquired the necessary functionality to protect against vulnerabilities and became available for deployment in the blockchain. Therefore, the next step will be to deploy a smart contract to the Ethereum blockchain and perform its core functions.

BLOCKCHAIN, SMART CONTRACT, VULNERABILITY, DEPLOY, AUDIT, TOKEN, CRYPTOCURRENCY, PROGRAMMING.

СПИСОК УМОВНИХ СКОРОЧЕНЬ

DeFi – Decentralized Finance;

DOS – Denial of Service;

ERC – Ethereum Request for Comments;

EVM – Ethereum Virtual Machine;

ICO – Initial Coin Offering;

KYS – Know Your Customer;

NFT – Non-fungible Token;

OWASP – Open Web Application Security Project;

PoW – Proof-of-work;

SHA – Secure Hash Algorithm.

ЗМІСТ

РОЗДІЛ 1. ВСТУП. СТАН ПИТАННЯ 8
1.1 Основна інформація про блокчейн 8
1.2 Основні криптовалюти та їх мережі 10
1.3 Блокчейн-експлорери 11
1.4 Смарт-контракти 13
1.5 Мова смарт-контрактів Solidity 18
1.6 Середовище розробки Remix 19
1.7 Огляд демонстраційного смарт-контракту 21
1.8 Висновок 23
РОЗДІЛ 2. ОГЛЯД ВРАЗЛИВОСТЕЙ СМАРТ-КОНТРАКТІВ 25
2.1 Повторний вхід 25
2.2 Код без наслідків 26
2.3 Контроль доступу 27
2.4 Відсутність перевірки значень, що повертаються функціями низького рівня 28
2.5 Арифметичні проблеми 29
2.6 Переупорядкування транзакцій 31
2.7 Відмова в обслуговуванні (DOS) 33
2.8 Ілюзія ентропії 36
2.9 Необроблені винятки 37
2.10 Неініціалізовані вказівники пам’яті 39
2.11 Автентифікація за допомогою tx.origin 40
2.12 Виклик delegatecall 42
2.13 Висновок 47
РОЗДІЛ 3. СПЕЦІАЛЬНА ЧАСТИНА 48
3.1 Проведення аудиту смарт-контракту PPSD.sol 48
3.2 Реалізація заходів щодо захисту смарт-контракту PPSD.sol від виявлених вразливостей 51

3.3 Висновок.. 61
РОЗДІЛ 4. ЕКОНОМІЧНИЙ РОЗДІЛ 62
4.1 Постановка задачі... 62
4.2 Виконання розрахунків 62
4.2.1 Розрахунок капітальних витрат 62
4.2.2 Розрахунок річних експлуатаційних витрат 65
4.2.3 Визначення річного економічного ефекту 66
4.2.4 Визначення та аналіз показників економічної ефективності...................... 68
4.3 Висновок.. 69
ВИСНОВКИ 70
ПЕРЕЛІК ПОСИЛАНЬ 72
ДОДАТОК А. Відомість матеріалів кваліфікаційної роботи ....
ДОДАТОК Б. Перелік документів на оптичному носії ....
ДОДАТОК В. Відгук керівника економічного розділу ....
ДОДАТОК Г. Відгук керівника..
ДОДАТОК Ґ. Лістинг коду смарт-контракту IERC20.sol.......................................
ДОДАТОК Д. Лістинг коду смарт-контракту ERC20.sol ....
ДОДАТОК Е. Початковий лістинг коду смарт-контракту PPSD.sol......................
ДОДАТОК Є. Лістинг коду смарт-контракту PPSD.sol з урахуванням запропонованих рішень щодо протидії виявленим вразливостям ....
ДОДАТОК Ж. Лістинг бібліотеки SafeMath.sol...

РОЗДІЛ 1. ВСТУП. СТАН ПИТАННЯ

1.1 Основна інформація про блокчейн

Перші згадки про блокчейн датуються 1991 роком, коли Стюарт Хабер та В. Скотт Сторнетта випустили свою роботу про криптографічно закріплений ланцюжок блоків. Вони хотіли впровадити систему, в якій часові мітки документів не можна було підробити. На основі цієї роботи у 2008 році Сатоші Накамото розробив перший блокчейн. Спочатку блокчейн служив як електронна книга для зберігання інформації про всі транзакції криптовалюти біткойн.

Мета блокчейну — дозволити записувати та поширювати цифрову інформацію, але не редагувати. Таким чином, блокчейн є основою для незмінних реєстрів або записів транзакцій, які не можна змінити, видалити або знищити.

Для того, щоб незнайомій з блокчейном людині зрозуміти, що він з себе представляє, структуру блокчейну можна порівняти з базою даних. Але порівняно з базою даних, блокчейн має декілька суттєвих відмінностей, таких як:

- Спосіб структури даних

  Класична база даних зберігає інформацію в таблицях, які можуть бути пов'язані одна з одною за допомогою унікального первинного ключа (в випадку якщо це реляційна база даних) або як прості пари «ключ - значення» (в випадку нереляційної бази даних). В свою чергу, блокчейн збирає інформацію у групи, також відомі як блоки, які містять набори інформації. Блоки мають установлену ємність і після заповнення блока інформацією, вони додаються до ланцюга з попередніх блоків, що і утворює назву «блокчейн». Кожен блок має свій унікальний ідентифікатор, криптографічний «хеш». Хеш не тільки захищає інформацію всередині блоку від будь-кого без необхідного коду, але також захищає місце блоку вздовж ланцюжка, ідентифікуючи блок, який був перед ним.

Криптографічний хеш — це набір цифр і букв, довжина яких є фіксованою та може складати до 64 цифр.

- Децентралізація

  Для того, щоб безпечно зберігати інформацію в базах даних та обслуговувати їх, компанії купують сервер або групу серверів, які знаходяться в одному приміщенні і підтримкою та обслуговуванням таких систем займаються конкретно призначені люди. Блокчейн також утворюється з багатьох серверів, але в даному випадку ними є комп'ютери користувачів, які знаходяться по всьому світу.

- Прозорість

  Через те, що блокчейн є децентралізованим, вся інформація про транзакції в ньому є відкритою та може бути переглянута будь-яким користувачем за допомогою спеціальних сайтів, які називаються блокчейн-експлорерами. Експлорери можуть показати таку інформацію про транзакції, як адреси відправника і отримувача в мережі, кількість відправленої валюти, номер блоку, хеш транзакції і т. д.

На даному етапі розвитку блокчейн активно порівнюють з банківською системою та навіть бачать в ньому заміну банкам в майбутньому. Це відбувається через те, що:

- Блокчейн, на відміну від банків, не має робочих і неробочих годин. Транзакції в ньому можуть проводитись в будь-який час.
- Комісія в банках є сталою і змінюється тільки при серйозних змінах в політиці банку або в економіці країни, в якій цей банк працює. В блокчейні комісія залежить від кількох чинників та часто коливається, тому користувачі можуть не проводити транзакцію, якщо вони бачать, що комісія надто висока, а зачекати поки вона знизиться.
- На виконання транзакцій в банках може йти до 3 робочих днів, а якщо користувач переводить велику суму, то йому доведеться чекати до 7

робочих днів. Транзакції в блокчейні проходять в середньому за 15 хвилин. При сильній завантаженості мережі, цей час може збільшитись до кількох годин.

- Для користування послугами банків, людина повинна пройти процедуру під назвою KYS. Ця процедура проводиться для верифікації людини і потребує документи, які підтверджують її особистість. Проводити транзакції в блокчейні може будь-хто, система не потребує ніяких даних про користувача до тих пір, поки він не захоче вивести свої активи на банківський рахунок. На цьому етапі криптобіржі також вимагають проходження процедури KYS.[1]

Блокчейн може використовуватись у різних сферах життєдіяльності. Наприклад, для відстеження шляху, який проходять продукти, для створення нової валюти, безпечного зберігання будь-якої документації, створення смарт-контрактів, забезпечення прозорості в голосуваннях та багато іншого.

1.2 Основні криптовалюти та їх мережі

За час існування блокчейну найбільш популярними криптовалютами стали Bitcoin (BTC), Ethereum (ETH), Binance Coin (BNB), Cardano (ADA), Tether (USDT). Більшість з них мають свої окремі блокчейни (мережі). В даній роботі будуть розглядатись вразливості смарт-контрактів в мережі Ethereum. Ethereum був створений в 2015 році канадцем російського походження Віталіком Бутеріним. Ця мережа має більш широкі амбіції, ніж біткойн, так як початково задумувалася не просто для запису транзакцій, а й для створення смарт-контрактів. Ethereum складається з основної мережі та щонайменше 4 тестових мереж. Основна мережа має назву Mainnet, тестові мережі, в свою чергу, називаються Ropsten, Kovan, Rinkeby та Goerly. Валюта, яка циркулює в тестових мережах не має ніякої цінності, ці мережі служать для тестування смарт-контрактів перед внесенням їх в основну мережу. Тестовий Ethereum можна отримати 1 раз на добу безплатно. За проведення трансляцій та інтеракцію зі смарт-контрактами користувачі повинні платити комісію, яка в даній системі має

назву газ. Величина комісії визначається майнерами, які встановлюють мінімальну ціну за свої послуги. Ці послуги полягають в наданні обчислювальної потужності своїх комп’ютерів для підтвердження транзакцій в мережі. Якщо ціна за послуги майнерів буде занадто низька, то вони можуть відмовлятись виконувати свою роботу.

1.3 Блокчейн-експлорери

Експлорери в блокчейні відіграють роль пошукової системи, яка зберігає в собі детальну інформацію про проведені в мережі транзакції. Про кожну транзакцію в мережі блокчейну експлорери можуть надавати наступну інформацію:

- Адресу гаманця відправника;
- Адресу гаманця отримувача;
- Назву валюти, яка передається;
- Кількість валюти, яка передається;
- Хеш транзакції;
- Номер блоку в блокчейні, в який записана транзакція;
- Час проведення транзакції.

В даній роботі транзакції будуть проводитись в тестовій мережі блокчейну Ethereum – Ropsten Test Network.

На рисунку 1.1 показаний приклад отримання інформації про транзакцію криптовалюти за допомогою експлорера.

Transaction Details

Overview Access List State

[This is a Ropsten **Testnet** transaction only]

Transaction Hash:	0xa57ce87ac1e919c11257b0a8233b114b827a1e6f6f5b0b3414c5b7453f8f7d37
Status:	Success
Block:	11391898 1 Block Confirmation
Timestamp:	1 min ago (Nov-09-2021 09:24:39 AM +UTC)
From:	0xcda0d6adcd0f1ccea6795f9b1f23a27ae643fe7c
To:	0xebfa3e2a81fda622d8be75d24f1944d154168c2d
Value:	0.3 Ether ($0.00)
Transaction Fee:	0.000052500000294 Ether ($0.00)
Gas Price:	0.000000002500000014 Ether (2.500000014 Gwei)
Txn Type:	2 (**EIP-1559**)

Рисунок 1.1 – Перегляд інформації про транзакцію в блокчейн-експлорері.

На рисунку 1.2 зображено приклад перегляду інформацію про створення смарт-контракту в експлорері.

Transaction Details

Overview Logs (2) Access List State

[This is a Ropsten **Testnet** transaction only]

Transaction Hash:	0x64dc19a1a8e0f8599da028040bdacda0d2006b3d328929ac3e450bcfba4dec49
Status:	Success
Block:	11392161 2 Block Confirmations
Timestamp:	45 secs ago (Nov-09-2021 10:31:46 AM +UTC)
From:	0xebfa3e2a81fda622d8be75d24f1944d154168c2d
Interacted With (To):	[Contract 0x215beefd76a33e165dcf417c63f1180b8630406f Created]
Tokens Transferred: 2	**From** 0x0000000000000... **To** 0xebfa3e2a81fda6... **For** 10,000,000,000 PPSD (PPSD) **From** 0x0000000000000... **To** 0x215beefd76a33... **For** 10,000,000,000 PPSD (PPSD)
Value:	0 Ether ($0.00)
Transaction Fee:	0.004778567517202843 Ether ($0.00)
Gas Price:	0.000000002500000009 Ether (2.500000009 Gwei)
Txn Type:	2 (**EIP-1559**)

Рисунок 1.2 – Перегляд інформації про створення смарт-контракту за допомогою блокчейн-експлорера.

Блокчейн-експлорери забезпечують одну із основних властивостей блокчейну - прозорість. За допомогою них можна отримувати інформацію про кожну транзакцію в мережі, а також слідкувати за переміщенням коштів при їх викраденні.

1.4 Смарт-контракти

Смарт-контракти – це програми, що зберігаються в блокчейні, які запускаються при дотриманні заздалегідь визначених умов. Зазвичай вони використовуються для автоматизації виконання угоди, щоб усі учасники могли бути негайно впевнені в результаті без участі посередника чи втрати часу. Вони також можуть автоматизувати робочий процес, запускаючи наступну дію, коли виконуються умови.

Смарт-контракти є потужною інфраструктурою для автоматизації, оскільки вони не контролюються центральним адміністратором і не є вразливими до окремих точок атаки з боку шкідливих об’єктів. Застосовані до багатосторонніх цифрових угод, програми смарт-контрактів можуть зменшити ризик контрагентів, підвищити ефективність, знизити витрати та забезпечити новий рівень прозорості процесів. Смарт-контракт точно визначає, як користувачі можуть взаємодіяти з ним, зокрема, хто може взаємодіяти зі смарт-контрактом, в який час і які вхідні дані призводять до яких результатів.

Смарт-контракти мають наступні переваги в порівнянні зі звичайними цифровими угодами:

- Безпека

  Виконання контракту на децентралізованій інфраструктурі блокчейн гарантує, що немає центральної точки невдачі в атаці, централізованого посередника для підкупу, а також жодного механізму, який будь-яка сторона чи центральний адміністратор могли б використовувати для підробки результату.
- Надійність

Надлишкова обробка та перевірка логіки контракту децентралізованою мережею вузлів забезпечує надійний захист від несанкціонованого доступу, час безперебійної роботи та коректність гарантує, що контракт буде виконано вчасно відповідно до його умов.

- Справедливість

  Використання децентралізованої мережі для розміщення та забезпечення виконання умов угоди зменшує можливість посередника, що має прибуток, використовувати свою привілейовану позицію для отримання ренти та вилучення вартості.

- Ефективність

  Автоматизація внутрішніх процесів угоди – депонування, обслуговування, виконання та/або розрахунки – означає, що жодній стороні не доведеться чекати введення даних вручну, контрагенту для виконання своїх зобов’язань або посереднику для обробки транзакції.

- Продуктивність

  Автономні смарт-контракти виконуються набагато швидше, ніж старомодний традиційний підхід. Оскільки всі параметри вже визначені в смарт-контрактах, їм потрібно лише зіставити їх перед тим, як він почне виконуватися.

- Безперервність

  Смарт-контракти безперервні. Це означає, що як тільки вони розпочинають своє виконання, їх не можна зупинити або перервати.

На даний момент смарт-контракти активно розвиваються та застосовуються в різних сферах життєдіяльності. Типові випадки застосування смарт-контрактів:

- Фінансові продукти (DeFi)

  Децентралізовані фінанси (DeFi) складаються з додатків, які використовують розумні контракти для відтворення традиційних фінансових продуктів і послуг, таких як грошові ринки, опціони,

стейблкоіни, біржі та управління активами, а також об’єднують кілька сервісів для створення нових фінансових примітивів за допомогою компонування без дозволу. Смарт-контракт може тримати кошти користувача на умовному депонуванні та розподіляти їх між користувачами на основі заздалегідь визначених умов. Наприклад, BarnBridge використовує смарт-контракти для автоматизації торгів для користувачів, які хочуть отримати доступ до основних засобів у парі цін (наприклад, 45% токен A, 55% токен B), a Aave використовує розумні контракти для полегшення кредитування та позики без дозволу та децентралізації.

- Блокчейн-ігри та розіграші

  Ігри на основі блокчейну використовують смарт-контракти для захищених від несанкціонованого виконання дій у грі. Криптоігри використовують технологію блокчейн одним із двох способів – або як основу всієї гри, або лише для її внутрішньоігрової економіки. Коли вся гра побудована на блокчейні, кожна взаємодія в грі зберігається і перевіряється як нові блоки. Усі ці ігри характеризуються своєю децентралізацією, особливо в порівнянні з традиційними відеоіграми. Одним із прикладів блокчейн-розіграшу є PoolTogether - програма, написана на блокчейні, де користувачі вносять свої кошти в загальний пул, який потім спрямовується на грошовий ринок (стейкінг), де він отримує відсотки. Після попередньо визначеного періоду часу розіграш закінчується, і переможець випадковим чином отримує всі нараховані відсотки, а всі інші можуть зняти свій початковий депозит і не втратити при цьому свої кошти.

- Страхування

  Параметричне страхування – це вид страхування, де виплата прив’язана безпосередньо до конкретної заздалегідь визначеної події. Розумні контракти забезпечують захищену від несанкціонованого доступу інфраструктуру для створення параметричних договорів страхування, які запускаються на основі введених даних. Наприклад, страхування врожаю

можна створити за допомогою розумних контрактів, коли користувач купує поліс на основі певної інформації про погоду, наприклад сезонних опадів у певному географічному місці. Наприкінці дії політики смарт-контракт автоматично видасть виплату, якщо кількість опадів у певному місці перевищує початкову вказану суму. Кінцеві користувачі не тільки отримують своєчасні виплати з меншими накладними витратами, але й сторона страхування може стати відкритою для громадськості за допомогою розумних контрактів. Розумний контракт дозволяє користувачам вносити кошти в пул, а потім розподіляє зібрані премії учасникам пулу на основі відсотка їхнього внеску в пул[2].

- Урядова система голосування

  Розумні контракти забезпечують безпечне середовище, що робить систему голосування менш сприйнятливою до маніпуляцій. Голосування за допомогою смарт-контрактів буде захищено реєстром, який надзвичайно важко розшифрувати. Більше того, смарт-контракти можуть збільшити плинність виборців, яка історично є низькою через неефективну систему, яка вимагає від виборців вишикуватися в чергу, демонструвати особистість та заповнювати форми. Голосування, перенесене онлайн за допомогою смарт-контрактів, може збільшити кількість учасників у системі голосування.

- Охорона здоров'я

  Блокчейн може зберігати закодовані медичні записи пацієнтів за допомогою приватного ключа. Лише окремим особам буде надано доступ до записів з міркувань конфіденційності. Так само дослідження можна проводити конфіденційно та безпечно за допомогою смарт-контрактів. Усі лікарняні квитанції пацієнтів можна зберігати на блокчейні та автоматично передавати їх страховим компаніям як підтвердження обслуговування. Крім того, реєстр можна використовувати для різних видів діяльності, таких як управління постачаннями, нагляд за ліками та дотримання нормативів.

- Ланцюг постачання

  Традиційно ланцюжки постачання страждають через паперові системи, де форми проходять кількома каналами для отримання схвалення. Трудомісткий процес підвищує ризик шахрайства та втрат. Блокчейн може звести нанівець такі ризики, надаючи доступну та безпечну цифрову версію сторонам, які беруть участь у ланцюжку. Смарт-контракти можна використовувати для управління запасами та автоматизації платежів і завдань.[3]

- Токени як валюта

  Токени в блокчейні являють собою окрему валюту, відмінну від основної валюти даної мережі. Ethereum дозволяє користувачам створювати смарт-контракти з будь-яким функціоналом, який не виходить за межі блокчейну, тому в мережі існує багато різновидів такої валюти. Це може бути стандартний токен, який дозволяє торгувати ним на біржах, може бути токен з додатковим функціоналом або взагалі криптобанк, який дозволяє купувати та продавати токени за допомогою всього одного смарт контракту.

- NFT-токени

  NFT – це токени, які використовуються для представлення власності на унікальні предмети. Вони дозволяють символізувати такі речі, як мистецтво, предмети колекціонування, нерухомість. Одночасно вони можуть мати лише одного офіційного власника, і вони захищені блокчейном – ніхто не може змінювати запис про право власності або копіювати/вставляти новий NFT.

  NFT означає незмінний токен. Незамінний – це економічний термін, який можна використовувати для опису таких речей, як меблі, файл пісні або комп'ютер. Ці речі не є взаємозамінними для інших предметів, оскільки вони мають унікальні властивості.[4]

1.5 Мова смарт-контрактів Solidity

Solidity – це об'єктно-орієнтована мова програмування високого рівня, яка використовується для створення смарт-контрактів. Після того, як мова була запропонована в 2014 році, вона була розроблена учасниками проекту Ethereum. Ця мова в основному використовується для створення смарт-контрактів на блокчейні Ethereum і на інших блокчейнух. Solidity схожа на одну з найпоширеніших мов програмування, JavaScript. Solidity також має схожі характеристики з мовами програмування C++ і Python.[5] Файли коду, написані на Solidity мають розширення “.sol”.

На рисунку 1.3 зображений приклад найпростішого смарт-контракту для створення токену в мережі Ethereum.

```
1   // SPDX-License-Identifier: GPL-3.0
2
3   pragma solidity ^0.8.0;
4
5   contract test_token {
6
7       mapping (address => uint256) public balances;
8
9       constructor(uint256 initialSupply) {
10          balances[msg.sender] = initialSupply;
11      }
12
13      function balanceOf(address account) external view returns (uint256) {
14          return balances[account];
15      }
16
17      function transfer (address to, uint256 amount) external {
18          require(to != address(0), "Cannot transfer to the zero address");
19          require(balanceOf(msg.sender) >= amount, "Transfer amount exceeds balance");
20          require(balanceOf(to) + amount >= balanceOf(to));
21          balances[msg.sender] -= amount;
22          balances[to] += amount;
23      }
24  }
25
```

Рисунок 1.3 – Приклад простого смарт-контракту для створення токена в блокчейні.

Ethereum дозволяє розробникам робити різноманітні смарт-контракти, але для того, щоб підтримувати взаємодію та коректну роботу продукту з гаманцями або біржами, вони повинні дотримуватись загальноприйнятих стандартів з написання смарт-контрактів. Стандарт являє собою набір функцій, які повинен

містити в собі смарт-контракт для того, щоб вважатися відповідним вибраному стандарту. Основними стандартами в блокчейні Ethereum є ERC-20, який застосовується при розробці токенів та ERC-721, який відноситься до NFT. Ці стандарти були реалізовані командою розробників під назвою OpenZeppelin. Ця команда розробила більшість стандартів та велику кількість периферійних смарт-контрактів, які починаючі блокчейн-розробники можуть підключати до своїх проектів без ризику зробити якусь критичну помилку в коді.

1.6 Середовище розробки Remix

Remix – це інтегроване середовище розробки смарт-контрактів, яке існує у вигляді веб-додатку та застосунку для персональних комп'ютерів та ноутбуків та працює на всіх найбільш популярних операційних системах, таких як Windows, MacOS та Linux. Він сприяє швидкому циклу розробки та має багатий набір плагінів з інтуїтивно зрозумілим графічним інтерфейсом. Remix використовується для всього шляху розробки контракту, воно має текстовий редактор коду, в якому пишуться смарт-контракти, а також плагіни для компіляції, перевірки на помилки в коді, статистичного аналізу коду, тестування та деплою контракту в блокчейн. На рисунку 1.4 приводиться головна сторінка веб-додатку, на якій можна створити новий смарт-контракт та скористатися навігацією по плагінам для подальшої роботи з ним.

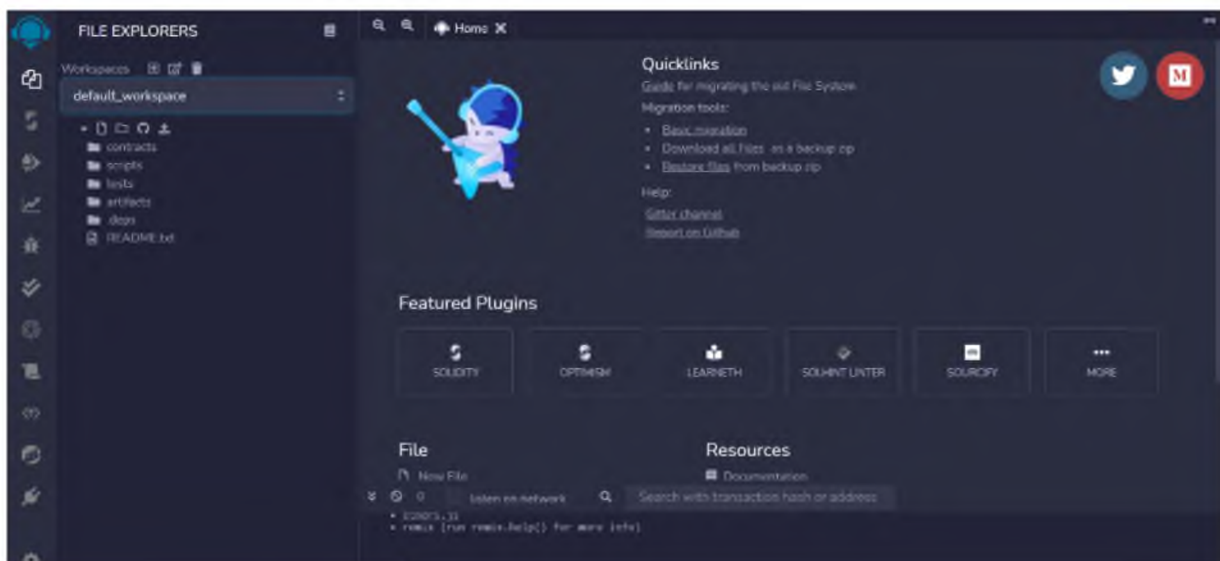

Рисунок 1.4 – Головна сторінка веб-додатку Remix.

На рисунку 1.5 ілюструється редактор смарт-контрактів та плагін компіляції, який дозволяє обирати мову програмування, а також версію компілятора для окремо взятого контракту.

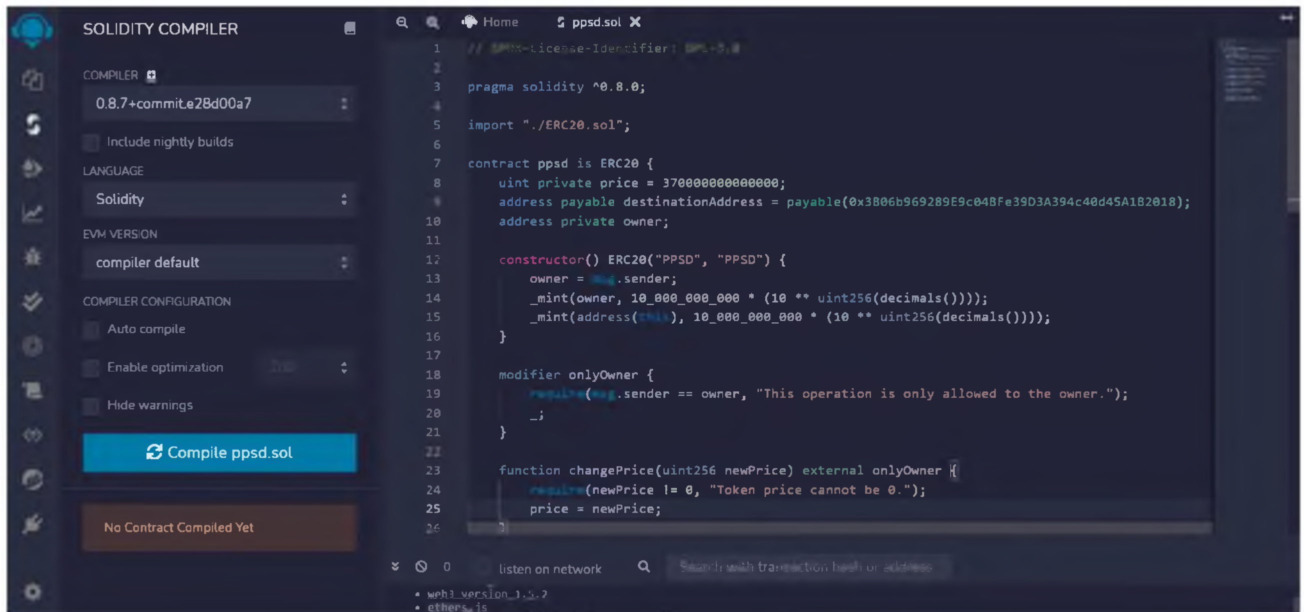

Рисунок 1.5 – Сторінка редагування та компіляції смарт-контракту.

На рисунку 1.6 зображено плагін, за допомогою якого можна виконати деплой смарт-контракту в обрану мережу блокчейну. Для цього на гаманці виконавця деплою повинна бути деяка кількість криптовалюти залежно від мережі, так як процес деплою смарт-контракту, як і будь-яка інтеракція з ним, являється транзакцією, за яку мережа бере комісію. Також даний плагін дозволяє звертатися до смарт-контрактів, деплой яких був проведений в поточній сесії браузера та взаємодіяти з ними шляхом виклику функцій смарт-контрактів.

Рисунок 1.6 – Демонстрація плагіну деплою та взаємодії зі смарт-контрактами.

1.7 Огляд демонстраційного смарт-контракту

Для наглядної демонстрації вразливостей смарт-контрактів в даній роботі, вони будуть показані на прикладі користувацького смарт-контракту, за допомогою якого було створено токен PPSD. В функції даного контракту також входить можливість покупки токенів користувачами шляхом відправки криптовалюти Ethereum на даний смарт-контракт. Криптовалюту, яку користувачі присилають на смарт-контракт автоматично відправляється на заздалегідь наданий замовником криптогаманець. Також власнику контракту надається можливість зміни курсу для покупки токену, зміни власника контракту та зміни адреси гаманця для пересилання криптовалюти Ethereum. Обраний смарт-контракт також є типовим прикладом стандарту Ethereum ERC-20. Структура коду даного смарт-контракту складається з 3 файлів, 2 з яких являють собою реалізацію стандарту ERC-20, третій файл коду наслідує два інші та реалізує додатковий функціонал токена. В таблиці 1.1 приведено перелік файлів та функцій, які вони виконують в даній структурі.

Таблиця 1.1 – Перелік файлів коду та їх призначення

Назва файлу	Функції, які він виконує
IERC20.sol	Являється інтерфейсом для смарт-контракту ERC20.sol та містить в собі перелік функцій, необхідних для відповідності стандарту ERC-20
ERC20.sol	Представляє собою реалізацію функцій стандарту ERC-20
PPSD.sol	Основний смарт-контракт. Наслідує функції з контракту ERC20.sol для створення токену PPSD та реалізує додатковий функціонал даного токену, пов'язаний з продажом токенів користувачам

В таблиці 1.2 приведений список функцій смарт-контракту ERC20.sol та їх призначення.

Таблиця 1.2 – Функції смарт-контракту ERC20.sol та їх пояснення

Назва функції	Призначення
_msgSender	Результатом є адреса гаманця ініціатора транзакції
name	Дає користувачеві назву токену
symbol	Показує скорочену назву токену
decimals	Показує кількість знаків за комою в кількості токену (дрібна доля)
totalSupply	Повертає загальну кількість випущених токенів
balanceOf	Демонструє баланс в токенах для конкретного користувача
transfer	Проводить передачу токенів з гаманця ініціатора транзакції на гаманець іншого користувача
allowance	Повертає кількість токенів одного користувача, які дозволено використовувати іншому користувачеві
approve	Встановлює кількість токенів одного користувача, які може використовувати інший користувач
transferFrom	Передає токени з одного гаманця на інший
increaseAllowance	Підвищує дозволену кількість використовуваних токенів іншого користувача

Продовження таблиці 1.2

Назва функції	Призначення
decreaseAllowance	Знижує кількість використовуваних токенів іншого користувача
_mint	Видає токени користувачеві, при цьому збільшує загальну кількість випущених токенів
_burn	Знищує певну кількість токенів користувача та зменшує загальну кількість випущених токенів

Таблиця 1.3 показує перелік функцій смарт-контракту PPSD.sol та їх пояснення.

Таблиця 1.3 – Функції смарт-контракту PPSD.sol та їх призначення

Назва функції	Призначення
changePrice	Змінює ціну токену
changeForwardingAddress	Змінює адресу гаманця для пересилки криптовалюти
changeOwner	Змінює власника смарт-контракту
retrieveTokens	Дозволяє власнику забрати залишок токенів, які знаходяться на балансі контракту
Receive	Дозволяє приймати криптовалюту від користувачів

Лістинг коду смарт-контракту IERC20.sol приведено в Додатку А, ERC20.sol в Додатку Б та PPSD.sol в Додатку В.

1.8 Висновок

В даному розділі було проведено ознайомлення з технологією блокчейн, показано основні сфери його застосування, а також продемонстровано його особливості та переваги порівняно з класичною банківською системою. Також було дано визначення смарт-контрактам, приведені приклади їх використання. Далі було розказано про комплект розробки смарт-контрактів, який містить в собі мову програмування Solidity, середовище розробки Remix та блокчейн-експлорер Etherscan. Теж було приведено лістинг та пояснення функцій демонстраційного смарт-контракту PPSD.sol. З цього можна зробити висновок, що смарт-контракти є важливою частиною блокчейну з фінансової точки зору, тому неправильне їх написання може завдати значних збитків власнику. Таким

чином проблема захисту смарт-контрактів від вразливостей являється однією з найбільш важливих та потребує особливої уваги при їх розробці.

РОЗДІЛ 2. ОГЛЯД ВРАЗЛИВОСТЕЙ СМАРТ-КОНТРАКТІВ

Щоденно через смарт-контракти проходять мільярди доларів. Тому криптовалюти, починаючи з часу свого виникнення, завжди були ціллю для шахраїв та хакерів. На даний момент існують спеціальні системи для аналізу смарт-контрактів на вразливості, розробники мови програмування Solidity постійно оновлюють її для запобігання найпоширеніших помилок розробників без досвіду. Але існує велика кількість вразливостей смарт-контрактів та варіантів атак на них. Подібні атаки можуть принести значних збитків для власників смарт-контрактів, так як часто токени, створені за допомогою даних контрактів служать як внутрішня валюта для блокчейн додатків та ігор і при втраті цих токенів, власники втратять також і аудиторію своїх блокчейн-додатків.

Нижче наведено список відомих атак, про які необхідно знати та брати до уваги під час створення захисту при написанні смарт-контрактів.

2.1 Повторний вхід

Однією з основних небезпек виклику зовнішніх контрактів є те, що вони можуть взяти на себе потік керування та внести зміни до даних, яких функція виклику не очікувала. Атака повторного входу в смарт-контракті є поширеною діяльністю. Ці атаки можуть повністю вичерпати кошти зі смарт-контракту. Атака повторного входу відбувається, коли функція здійснює зовнішній виклик до іншого ненадійного контракту. Потім ненадійний контракт робить рекурсивний виклик до вихідної функції, намагаючись вичерпати кошти. Простим прикладом є те, що контракт виконує певні дії всередині функції із змінною балансу та відкриває функцію виведення коштів. Якщо вразливий контракт передає кошти до того, як він встановлює баланс на нуль, зловмисник може рекурсивно викликати функцію зняття коштів повторно і вичерпати весь контракт. На рисунку 2.1 показаний приклад ненадійного смарт-контракту, який може піддатися атаці з повторним входом.[6]

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
}
```

Рисунок 2.1 – Приклад вразливого до повторного входу смарт-контракту.

2.2 Код без наслідків

У Solidity можна написати код, який не дає очікуваних ефектів. Наразі компілятор Solidity не повертає попередження для коду без ефектів. Така непомітна помилка розробника може призвести до введення «мертвого» коду, який не виконує належним чином заплановану дію.

Наприклад, у рядку коду msg.sender.call.value(address(this).balance)(""); легко пропустити кінцеві дужки, що може призвести до виконання функції без переказу коштів на msg.sender.

На рисунку 2.2 демонструється приклад такого смарт-контракту, результатом якого буде те, що за виконання транзакції буде зніматись комісія з її ініціатора, але свої функції даний контракт виконувати не буде.[7]

```
pragma solidity ^0.5.0;

contract Wallet {
    mapping(address => uint) balance;

    // Deposit funds in contract
    function deposit(uint amount) public payable {
        require(msg.value == amount, 'msg.value must be equal to amount');
        balance[msg.sender] = amount;
    }

    // Withdraw funds from contract
    function withdraw(uint amount) public {
        require(amount <= balance[msg.sender], 'amount must be less than balance'

        uint previousBalance = balance[msg.sender];
        balance[msg.sender] = previousBalance - amount;

        // Attempt to send amount from the contract to msg.sender
        msg.sender.call.value(amount);
    }
}
```

Рисунок 2.2 – Приклад смарт-контракту з неявною помилкою в коді.

2.3 Контроль доступу

Проблеми з контролем доступу поширені в усіх програмах, а не тільки в смарт-контрактах. Фактично, це 1 місце в топ-10 OWASP в 2021 році.[8] Зазвичай доступ до функціональних можливостей контракту здійснюється через його публічні або зовнішні функції. Хоча незахищені налаштування видимості дають зловмисникам прості способи отримати доступ до приватних значень або логіки контракту, обхід контролю доступу іноді є більш тонким. Ці вразливості можуть виникнути, коли контракти використовують застарілий tx.origin для перевірки абонентів або обробляють велику логіку авторизації з тривалими вимогами. Прикладом даної вразливості може служити смарт-контракт, приведений на рисунку 2.3. Даний контракт містить в собі функцію зміни власника. Але так як дана функція є незахищеною від доступу для сторонніх осіб, будь-який користувач може викликати дану функцію та встановити замість адреси власника смарт-контракту свою власну адресу. Такі дії можуть значною

мірою вплинути на логіку та подальшу роботу такого смарт-контракту, а також завдати значних збитків оригінальному власникові смарт-контракту.[9]

```
1   // SPDX-License-Identifier: GPL-3.0
2
3   pragma solidity ^0.8.0;
4
5   contract test_contract {
6
7       address private owner;
8
9       constructor() {
10          owner = msg.sender;
11      }
12
13      function changeOwner(address newOwner) public {
14          owner = newOwner;
15      }
16  }
17
```

Рисунок 2.3 – Приклад смарт-контракту без контролю доступу.

2.4 Відсутність перевірки значень, що повертаються функціями низького рівня

Однією з найглибших особливостей Solidity є функції низького рівня call(), callcode(), delegatecall() і send(). Їхня поведінка під час виникнення помилок суттєво відрізняється від інших функцій Solidity, оскільки вони не призведуть до повної реверсії поточного виконання. Замість цього вони повернуть логічне значення, встановлене на false, і код продовжить працювати. Це може здивувати розробників і, якщо значення, що повертається для таких низькорівневих викликів, не перевіряється, може призвести до небажаних результатів. Код смарт-контракту, приведений на рисунку 2.4 є прикладом того, що може піти не так, якщо забути перевірити значення, яке повертає функція send(). Якщо виклик

використовується для відправки Ethereum до смарт-контракту, який його не приймає (наприклад, тому що контракт-отримувач не має відповідної функції для цього), функція send() поверне значення false. Оскільки в даному прикладі значення, що повертається, не перевіряється, зміни контракту, які проводить дана функція, не будуть скасовані, і змінна etherLeft в кінцевому підсумку матиме неправильне значення.[10]

```
function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

Рисунок 2.4 – Приклад смарт-контракту без перевірки значення, яке повертає функція send().

2.5 Арифметичні проблеми

Віртуальна машина Ethereum (EVM) визначає типи даних фіксованого розміру для цілих чисел. Це означає, що ціла змінна має лише певний діапазон чисел, які вона може представляти. Наприклад, uint8 може зберігати лише числа в діапазоні [0, 255]. Спроба зберегти 256 в uint8 призведе до переповнення та поміщення в змінну значення 0. Якщо не звертати увагу на дану вразливість, змінні смарт-контракту можуть бути використані зловмисниками при відсутності перевірки введених користувачем даних. В результаті цього будуть виконані обчислення, які призводять до того, що в змінні потрапляють числа, які лежать за межами діапазону типу даних, який їх зберігає.

Наприклад, віднімання 1 від змінної типу даних uint8 (ціле число без знаку з 8 біт, тобто тільки позитивне), яка зберігає 0 як своє значення, призведе до

числа 255. Цій змінній було призначено число нижче діапазону uint8, тому результат обертається і дає найбільше число, яке може зберігати uint8. Аналогічно, додавання 2^8=256 до uint8 залишить змінну незмінною, оскільки було обгорнуто всю довжину uint (в математиці це схоже на додавання 2π до кута тригонометричної функції, sin(x) = sin(x+2π)). Додавання чисел, більших за діапазон типу даних, називається переповненням. Для наочності, додавання 257 до uint8, яке на даний момент має нульове значення, призведе до числа 1. Змінні фіксованого типу є циклічними, тому при додаванні до змінної числа, яке перевищує ліміт типу даних для даної змінної, число перетвориться на 0 та почне відлік заново. Так само якщо відняти від такої змінної число, яке більше ніж вона сама, то значення змінної перейде в верхню межу типу даних і буде віднімати від 256.

Ця вразливість дозволяє зловмисникам неправильно використовувати код і створювати несподівані логічні потоки. На рисунку 2.5 показаний смарт-контракт, який є вразливим до арифметичного переповнення. Цей смарт-контракт розроблений як тимчасове сховище, в яке користувачі можуть внести Ethereum, і він буде заблокований там щонайменше на тиждень. За бажанням користувач може продовжити час блокування до 1 тижня. Після внесення депозиту користувач буде впевнений, що його криптовалюта безпечно заблокована принаймні на тиждень. Але в представленому смарт-контракті не виконується перевірка арифметичних операцій на переповнення, тому у випадку, якщо користувач втратить свій закритий ключ, зловмисник міг би використати переповнення для отримання ефіру, незалежно від часу блокування. Зловмисник може визначити поточний час блокування для адреси, до якої він має ключ (це є відкритою змінною). Її назва – userLockTime. Потім він може викликати функцію збільшення часу блокування криптовалюти LockTime і передати як аргумент число 2^256 - userLockTime. Це число буде додано до поточного userLockTime і спричинить переповнення, скинувши lockTime[msg.sender] на 0. Потім

зловмисник може просто викликати функцію вилучення, щоб отримати свою винагороду, так як криптовалюта буде повністю розблокована.[11]

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        uint transferValue = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

Рисунок 2.5 – Приклад смарт-контракту, який є вразливим до арифметичного переповнення.

2.6 Переупорядкування транзакцій

Як і більшість блокчейнів, вузли Ethereum об'єднують транзакції та формують їх у блоки. Транзакції вважаються дійсними лише після того, як майнер вирішив механізм консенсусу (наразі ETHASH PoW для Ethereum). Майнер, який розв'язує блок, також вибирає, які транзакції з пулу будуть включені в блок, зазвичай це впорядковується за кількістю газу, яка буда внесена за дану транзакцію. Тут лежить потенційний вектор атаки. Зловмисник може

спостерігати в пулі транзакцій на предмет транзакцій, які можуть містити рішення проблем або змінювати стан контракту, який є небажаним для зловмисника. Потім зловмисник може отримати дані з цієї транзакції та створити власну транзакцію з вищою кількістю газу, заплачену за транзакцію та включити свою транзакцію в блок перед оригінальною. На рисунку 2.6 приводиться приклад простого смарт-контракту, логіка якого складається з того, що користувач повинен вгадати слово, яке перетворено в хеш за допомогою функції sha3. Користувач, який зможе знайти попереднє зображення хешу sha3 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a, може надіслати рішення та отримати 1000 Eth. Якщо один із користувачів зрозумів яке рішення є правильним, він може викликати функцію solve() зі своєю відповіддю як параметр. Зловмисники, в свою чергу, можуть стежити за пулом транзакцій для тих, хто подає рішення. Вони бачать це рішення, перевіряють його дійсність, а потім подають еквівалентну транзакцію з набагато вищою ціною на газ, ніж початкова транзакція. Майнер, який розв'язує блок, швидше за все, віддасть перевагу зловмиснику через вищу ціну на газ і прийме його транзакцію раніше вихідного розв'язувача. Зловмисник забере 1000 ефіру, а користувач, який вирішив проблему першим, нічого не отримає так як на балансі смарт-контракту не залишиться Ethereum.[12]

```
1   contract FindThisHash {
2
3       bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;
4
5       constructor() public payable {}
6
7       function solve(string solution) public {
8           // If you can find the pre image of the hash, receive 1000 ether
9           require(hash == sha3(solution));
10          msg.sender.transfer(1000 ether);
11      }
12  }
```

Рисунок 2.6 – Приклад смарт-контракту, який може стати для атаки за допомогою переупорядкування транзакцій.

2.7 Відмова в обслуговуванні (DOS)

Ця категорія дуже широка, але в основному складається з атак, коли користувачі можуть залишити контракт недіючим на невеликий період часу, а в деяких випадках і назавжди. Це може назавжди заблокувати криптовалюту у цих контрактах. Існують різні способи, через які контракт може стати недійсним. Далі буде приведено деякі потенційно менш очевидні шаблони кодування Solidity із нюансами блокчейну, які можуть призвести до здійснення атаки DOS зловмисниками.

2.7.1. Зовнішні виклики без платні за газ.

Зловмисник може здійснити зовнішній виклик до невідомого контракту та продовжити обробку транзакції незалежно від того, чи не завершиться цей виклик. Зазвичай це досягається за допомогою виклику операції CALL, яка не скасовує транзакцію, якщо виклик не вдається. На рисунку 2.7 показаний приклад смарт-контракту, який являється гаманцем для криптовалюти та повільно повертає Ethereum, коли викликається функція withdraw(). Користувач може додати свою адресу та витратити газ, щоб викликати функцію виведення, надаючи як користувачеві, так і власнику даного смарт-контракту 1% від загального балансу контракту.

```
1   contract TrickleWallet {
2
3       address public partner; // withdrawal partner - pay the gas, split the withdraw
4       address public constant owner = 0xA9E;
5       uint timeLastWithdrawn;
6       mapping(address => uint) withdrawPartnerBalances; // keep track of partners balances
7
8       function setWithdrawPartner(address _partner) public {
9           require(partner == '0x0' || msg.sender == partner);
10          partner = _partner;
11      }
12
13      // withdraw 1% to recipient and 1% to owner
14      function withdraw() public {
15          uint amountToSend = address(this).balance / 100;
16          // perform a call without checking return
17          // the recipient can revert, the owner will still get their share
18          partner.call.value(amountToSend)();
19          owner.transfer(amountToSend);
20          // keep track of last withdrawal time
21          timeLastWithdrawn = now;
22          withdrawPartnerBalances[partner] += amountToSend;
23      }
24
25      // allow deposit of funds
26      function() payable {}
27
28      // convenience function
29      function contractBalance() view returns (uint) {
30          return address(this).balance;
31      }
32  }
```

Рисунок 2.7 – Приклад смарт-контракту, який може піддатися відмові в обслуговуванні.

В даному смарт-контракті можна побачити, що в лінії виконується зовнішній виклик, надсилаючи 1% від балансу контракту на вказаний користувачем обліковий запис. Причина використання коду операції CALL полягає в тому, щоб переконатися, що власник все ще отримує гроші, навіть якщо зовнішній виклик повертається. Проблема полягає в тому, що транзакція надішле весь свій газ на зовнішній виклик. Якщо користувач був зловмисним, він міг би створити контракт, який споживав би весь газ і змусив усі транзакції withdraw() завершити невдачу через нестачу газу. Рисунок 2.8 показує, як може виглядати шкідливий смарт-контракт, який споживає весь газ смарт-контракту, показаного на рисунку 2.7.

```
contract ConsumeAllGas {
    function () payable {
        // an assert consumes all transaction gas, unlike a
        //revert which returns the remaining gas
        assert(1==2);
    }
}
```

Рисунок 2.8. Смарт-контракт, який може бути використаний для зпалювання всього газу контракту-жертви.

Якщо користувач вирішує, що йому не подобається власник контракту, він може встановити адресу цього смарт-контракту в змінну partner контракту TrickleWallet та назавжди заблокувати всі кошти в початковому смарт-контракті.

Щоб запобігти таким векторам атаки DOS, при розробці смарт-контракту треба переконатись, що в зовнішньому виклику вказано максимальну кількість газу, яку може використати дана транзакція. Для того, щоб виправити дану вразливість в смарт-контракті, який показано на рисунку 2.7, необхідно змінити рядок 17 таким чином, щоб вийшов наступний код: partner.call.gas(50000).value(amountToSend)(); Ця модифікація дозволяє витратити лише 50 000 газу на зовнішню транзакцію. Власник може встановити ціну на газ, більшу за цю, для того, щоб транзакція була завершена, незалежно від того, скільки зовнішня транзакція використовує.

2.7.2 Прохід в циклі по маппінгам або масивам, якими маніпулюють ззовні

В мові програмування смарт-контрактів Solidity тип даних mapping містить в собі дані типу ключ – значення. Він має схожу структуру зі словниками з інших традиційних мов програмування. В смарт-контрактах даний тип частіш за все використовується для зберігання балансу токенів для користувачів або інших їх даних.

Існують різні форми такого роду паттернів. Зазвичай це проявляється в сценаріях, коли власник хоче розподілити токени між своїми інвесторами, і робить це за допомогою функції, подібної на distribute(), яку можна побачити на рисунку 2.9.

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of tokens to the address "to"
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

Рисунок 2.9 – Приклад смарт-контракту, який місіть в собі цикл.

Цикл у цьому контракті виконується над масивом, який можна штучно роздути. Зловмисник може створити багато облікових записів користувачів з мінімальним балансом, що робить масив інвесторів великим. Це може призвести до того, що газ, необхідний для виконання циклу for, перевищить ліміт газу блоку, тим самим зробить функцію distribute() непрацездатною.[13] Через вірогідність проведення такої атаки, а також велику кількість газу для їх виконання, цикли не рекомендується використовувати в смарт-контрактах взагалі.

2.8 Ілюзія ентропії

Усі транзакції в блокчейні Ethereum є детермінованими операціями переходу стану. Це означає, що кожна транзакція змінює глобальний стан екосистеми Ethereum і робить це обчислюваним способом без невизначеності. Зрештою, це означає, що всередині екосистеми блокчейн немає джерела ентропії

чи випадковості. У Solidity немає функції rand(), яка повертає випадкове значення та яка є в більшості високорівневих мовах програмування. Деякі з перших контрактів, створених на платформі Ethereum, були засновані на азартних іграх. По суті, азартні ігри вимагають невизначеності (щось, на що можна зробити ставку), що ускладнює створення системи азартних ігор на блокчейні. Невизначеність повинна виходити з джерела, зовнішнього по відношенню до блокчейну. Це тяжко зробити в блокчейні коли неохідна справді випадкова величина. Поширеною помилкою є використання майбутніх змінних блоку, таких як хеші, часові позначки, номер блоку або ліміт газу. Проблема з ними полягає в тому, що вони контролюються майнером, який видобуває блок, і тому вони не є насправді випадковими. Наприклад, смарт-контракт, який виконує функції рулетки, повертає чорне число, якщо хеш наступного блоку закінчується парним числом. Майнер може поставити 1 мільйон доларів на чорне, якщо він розв’яже наступний блок і зрозуміє, що хеш закінчується непарним числом. Таким чином майнер може не опубліковувати свій блок і не видобувати інший, доки не знайде рішення, де хеш блоку буде парним числом. Використання змінних минулого або теперішнього часу може бути ще більш руйнівним. Крім того, використання виключно блокових змінних означає, що псевдовипадкове число буде однаковим для всіх транзакцій у блоці, тому зловмисник може примножити свої виграші, виконавши багато транзакцій у блоці.[14]

2.9 Необроблені винятки

Solidity, як і усі найбільш популярні мови програмування, має функціонал для відстеження та оброблення помилок при виконанні функцій або викликів. При розробці функцій, які змінюють стан смарт-контракту або його змінні, необхідно ретельно продумувати всі можливі сценарії неправильного або зловмисного виклику даної функції. Якщо смарт-контракт не перевіряє вхідні параметри таких функцій, зловмисник може скористатися цим, щоб нашкодити такому смарт-контракту або викрасти з нього криптовалюту. На рисунку 2.10

приведено приклад смарт-контракту, який являється банком токенів. Користувачі можуть купувати токени шляхом відправки криптовалюти Ethereum на даний смарт-контракт та отримувати кількість токенів еквівалентну кількості надісланого Ethereum помноженому на значення змінної priceMultiplier.

```
1  // SPDX-License-Identifier: GPL-3.0
2  
3  pragma solidity ^0.8.0;
4  
5  import "./ERC20.sol";
6  
7  contract exceptionsTest {
8      uint256 public priceMultiplier = 100;
9  
10     constructor() ERC20("token", "token") {
11         _mint(address(this), 10_000_000 * (10 ** uint256(decimals())));
12     }
13 
14     function changeMultiplier(uint256 newMult) external {
15         priceMultiplier = newMult;
16     }
17 
18     receive() external payable {
19         uint256 amount = msg.value * priceMultiplier;
20         _transfer(address(this), msg.sender, amount);
21     }
22 }
```

Рисунок 2.10 – Прикла смарт-контракту без відстежування помилок.

В рядку 14 даного контракту реалізована функція зміни множника для розрахунку кількості отримуваних користувачем токенів. Вона приймає в якості параметра нове значення множника. Але так як в даній функції не проводиться перевірка на помилкове значення параметру, зловмисник може викликати дану функцію і вказати в якості параметра значення 1000000 та відправити Ethereum на даний смарт-контракт. Таким чином зловмисник зможе отримати набагато більшу кількусть токенів, ніж було зазначено власником смарт-контракту.

2.10 Неініціалізовані вказівники пам’яті

EVM зберігає значення змінних в типах даних, таких як memory або storage. Під час розробки смарт-контрактів необхідно розуміти, як це робиться та необхідні типи даних за замовчуванням для локальних змінних функцій. Це пов’язано з тим, що можна створювати вразливі контракти шляхом неналежної ініціалізації змінних. На рисунку 2.11 продемонстровано простий смарт-контракт, який здійнює роботу реєстратора імен та містить лише одну функцію. Коли контракт розблоковано, він дозволяє будь-кому зареєструвати ім’я як хеш bytes32 і зіставити це ім’я з адресою. На жаль, цей реєстратор спочатку заблоковано, і вимога в рядку 23 не дозволяє функції register() додавати записи імен. Однак у цьому контракті є вразливість, яка дозволяє реєструвати ім’я незалежно від розблокованої змінної.

```
1   contract NameRegistrar {
2
3       bool public unlocked = false;  // registrar locked, no name updates
4
5       struct NameRecord { // map hashes to addresses
6           bytes32 name;
7           address mappedAddress;
8       }
9
10      mapping(address => NameRecord) public registeredNameRecord; // records who registered names
11      mapping(bytes32 => address) public resolve; // resolves hashes to addresses
12
13      function register(bytes32 _name, address _mappedAddress) public {
14          // set up the new NameRecord
15          NameRecord newRecord;
16          newRecord.name = _name;
17          newRecord.mappedAddress = _mappedAddress;
18
19          resolve[_name] = _mappedAddress;
20          registeredNameRecord[msg.sender] = newRecord;
21
22          require(unlocked); // only allow registrations if contract is unlocked
23      }
24  }
```

Рисунок 2.11 – Демонстрація смарт-контракту з неправильною ініціалізацією змінних.

Щоб зрозуміти як працює дана вразливість, спочатку потрібно зрозуміти, як працює пам’ять в Solidity. Змінні стану зберігаються послідовно в слотах, в тому порядку як вони з’являються в смарт-контракті. Таким чином, змінна

unlocked існує в слоті 0, змінна registeredNameRecord існує в слоті 1, а змінна resolve – в слоті 2 і т. д. Кожен із цих слотів має розмір 32 байт. Змінна unlocked в двійковому коді буде виглядати як 0x000...0 (64 нулі, за винятком 0x) для значення false або 0x000...1 (63 нулі) для true. З цього можна зробити висновок, що у цьому конкретному прикладі є значний надлишок займаної змінною пам’яті. Solidity за замовчуванням зберігає складні типи даних, такі як структури, під час ініціалізації їх як локальних змінних. Таким чином, змінна newRecord у рядку 16 за замовчуванням зберігається в пам’яті. Уразливість викликана тим, що при об’яві змінної newRecord її не було ініціалізовано. Оскільки за замовчуванням вона знаходиться в пам’яті, вона стає вказівником на пам’ять, а оскільки вона не ініціалізована, змінна вказує на слот 0, в якому зберігається змінна unlocked. Далі в рядках 17 і 18 змінна nameRecord.name встановлюємо на _name і nameRecord.mappedAddress на _mappedAddress, це фактично змінює місце зберігання слота 0 і слота 1, що змінює як змінну unlocked, так і слот пам’яті, пов’язаний з registeredNameRecord. Це означає, що unlocked можна безпосередньо змінити, просто за допомогою параметра _name функції register(). Тому, якщо останній байт змінної _name буде відмінний від нуля, він змінить останній байт слота пам’яті 0 і безпосередньо змінить unlocked на true. Такі значення _name пройдуть перевірку require() у рядку 23, оскільки значення unlocked було встановлено на true.[15]

2.11 Автентифікація за допомогою tx.origin

Solidity має глобальну змінну під назвою tx.origin, яка проходить по всьому стеку викликів і повертає адресу облікового запису користуваяа, який спочатку надіслав транзакцію. Використання цієї змінної для аутентифікації в смарт-контрактах робить контракт вразливим до фішингової атаки. Такі атаки можуть обманом змусити користувачів виконати автентифіковані дії щодо вразливого контракту. Рисунок 2.12 демонструє приклад смарт-контракту, який може бути вразливим для фішингу через використання tx.origin для автентифікації користувача.

```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Рисунок 2.12 – Приклад вразливого до фішингу смарт-контракту.

В рядку 11 даного смарт-контракту відбувається авторизація користувача за допомогою функції withdrawAll() за допомогою виклику tx.origin. Цей контракт дозволяє зловмиснику створити атакуючий контракт, який показаний на рисунку 2.13.

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attackers address to receive funds.

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}
```

Рисунок 2.13 – Приклад смарт-контракту для фішингової атаки.

Щоб використати цей контракт, зловмисник виконає його деплой, а потім переконає власника контракту Phishable надіслати цьому контракту певну

кількість ефіру. Зловмисник може замаскувати цей контракт під власну приватну адресу, та обманути жертву, щоб та надіслала певну форму транзакції на цю адресу. Жертва, якщо не буде обережною, може не помітити, що на адресі зловмисника є код, або зловмисник може видати його за гаманець із кількома підписами чи деякий розширений гаманець для зберігання даних. Це може відбутися, тому що вихідний код публічних контрактів недоступний за замовчуванням, а стає доступним тільки після його верифікації власником в блокчейн-експлорері. У будь-якому випадку, якщо жертва надішле транзакцію з достатньою кількістю газу на адресу AttackContract, ця транзакція викличе функцію, яка, у свою чергу, викликає функцію withdrawAll() контракту Phishable, в якій в якості параметра вказана адреса атакуючого смарт-контракту. Це призведе до виведення всіх коштів з контракту Phishable на адресу зловмисника. Це відбудеться тому, що адреса, яка ініціалізувала виклик, була власником контракту Phishable. Таким чином, при перевірці адреси, tx.origin буде дорівнювати адресі власника, і перевірка в рядку 11 смарт-контракту Phishable пройде успішно.

2.12 Виклик delegatecall

Операційні коди call і delegatecall можуть бути корисні для того, щоб дозволити розробникам смарт-контрактів Ethereum модулювати свій код. Стандартні зовнішні виклики повідомлень до контрактів обробляються кодом операції CALL, за допомогою якого код запускається в контексті зовнішнього контракту/функції. Код операції delegatecall ідентичний стандартному виклику повідомлення, за винятком того, що код, який виконується за цільовою адресою, виконується в контексті контракту виклику разом із тим фактом, що msg.sender і msg.value залишаються незмінними. Ця функція дозволяє реалізувати бібліотеки, за допомогою яких розробники можуть створювати повторно використовуваний код для майбутніх контрактів. Відмінності між цими двома кодами операції прості та інтуїтивно зрозумілі, але використання delegatecall може призвести до несподіваного виконання коду. Природа delegatecall, що зберігає контекст,

довела, що створення власних бібліотек без уразливостей не так просто, як могло б здатися на перший погляд. Код у самих бібліотеках може бути безпечним і вільним від вразливостей, однак під час запуску в контексті іншої програми можуть виникнути нові вразливості. На рисунку 2.14 зображений смарт-контракт, який може служити прикладом цього. Даний контракт являється бібліотекою та працює з числами Фібоначчі.

```
1   // library contract - calculates fibonacci-like numbers;
2   contract FibonacciLib {
3       // initializing the standard fibonacci sequence;
4       uint public start;
5       uint public calculatedFibNumber;
6
7       // modify the zeroth number in the sequence
8       function setStart(uint _start) public {
9           start = _start;
10      }
11
12      function setFibonacci(uint n) public {
13          calculatedFibNumber = fibonacci(n);
14      }
15
16      function fibonacci(uint n) internal returns (uint) {
17          if (n == 0) return start;
18          else if (n == 1) return start + 1;
19          else return fibonacci(n - 1) + fibonacci(n - 2);
20      }
21  }
```

Рисунок 2.14 – Приклад смарт-контракту, який являється бібліотекою.

Ця бібліотека забезпечує функцію, яка може генерувати n-число Фібоначчі в послідовності. Це дозволяє користувачам змінювати початковий номер послідовності і обчислювати n-числа, подібні до Фібоначчі в цій новій послідовності. На рисунку 2.15 показаний смарт-контракт, який використовує цю бібліотеку з рисунку 2.14.

```
contract FibonacciBalance {

    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the fibonancci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    // constructor - loads the contract with ether
    constructor(address _fibonacciLibrary) public payable {
        fibonacciLibrary = _fibonacciLibrary;
    }

    function withdraw() {
        withdrawalCounter += 1;
        // calculate the fibonacci number for the current withdrawal user
        // this sets calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
    }

    // allow users to call fibonacci library functions
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

Рисунок 2.15 – Смарт-контракт, який використовує бібліотеку чисел Фібоначчі.

Даний контракт дозволяє учаснику отримати Ethereum з контракту, при цьому кількість ефіру дорівнюватиме числу Фібоначчі, що відповідає номеру в черзі кожного учасника, тобто перший учасник отримує 1 Ether, другий також отримує 1, третій отримує 2, четвертий отримує 3, п’ятий 5 і так далі доки залишок контракту не стане меншим за число Фібоначчі, яке визначається. У цьому смарт-контракті є змінна під назвою fibSig. Вона містить перші 4 байти хешу Keccak (SHA-3) рядка "setFibonacci(uint256)". Така структура змінної відома як селектор функцій і поміщається в calldata, щоб указати, яка функція смарт-контракту буде викликана. Даний селектор використовується у функції delegatecall у рядку 21, щоб указати, що буде викликана функція

setFibonacci(uint256). Другим аргументом у delegatecall є параметр, який передається даній функції.

В даному смарт-контракту змінна стану start використовується як в бібліотеці, так і в основному контракті виклику. У контракті бібліотеки start використовується для вказівки початку послідовності Фібоначчі і встановлюється на 0, тоді як у контракті FibonacciBalance — на 3. Також резервна функція в контракті FibonacciBalance дозволяє передавати всі виклики до контракту бібліотеки, що дозволяє також викликати функцію setStart() бібліотечного смарт-контракту. Згадуючи, що стан контракту зберігається, може з'явитися можливість змінити стан початкової змінної в локальному контракті FibonnacciBalance. В такому разі, це дозволить вилучити більше ефіру, оскільки отриманий обчислений FibNumber залежить від початкової змінної (як показано в контракті бібліотеки). Насправді функція setStart() не змінює і не може змінювати початкову змінну в контракті FibonacciBalance. Основна вразливість у цьому контракті значно гірша, ніж просто зміна початкової змінної. Змінні стану або зберігання розміщуються в слотах пам'яті послідовно, як вони вводяться в контракт. Смарт-контракт, який виконує функції бібліотеки має дві змінні стану: start і calculatedFibNumber. Перша змінна – це start, тому вона зберігається в пам'яті смарт-контракту в слоті 0. Друга змінна, calculatedFibNumber, поміщається в наступний доступний слот зберігання, слот 1. Функція setStart() приймає вхідні дані і встановлює для початку будь-який вхідний параметр. Таким чином, ця функція встановлює слот пам'яті 0 на будь-який вхід, який приймає функція setStart(). Аналогічно, функція setFibonacci() встановлює для змінної calculatedFibNumber результат fibonacci(n).

Слот пам'яті 0 смарт-контракту FibonacciBalance відповідає адресі fibonacciLibrary, а слот 1 відповідає обчисленому FibNumber. Саме в цьому неправильному відображенні виникає вразливість. delegatecall зберігає контекст контракту. Це означає, що код, який виконується через delegatecall, буде діяти на стан контракту, що його викликає. В функції withdraw() в рядку 21 смарт-

контракту FibonacciBalance, який зображений на рисунку 2.15 виконується виклик fibonacciLibrary.delegatecall(fibSig, withdrawalCounter). Дана дія проводить виклик функції setFibonacci(), яка змінює слот пам'яті 1, який в даному випадку зберігає змінну calculatedFibNumber. Після виконання даної процедури змінна cdlculatedFibNumber коригується. Однак, початкова змінна в контракті FibonacciLib розташована в слоті пам'яті 0, який є адресою смарт-контракту fibonacciLibrary в поточному контракті. Це означає, що функція fibonacci() дасть несподіваний результат. Це відбудеться тому, що він посилається на початковий вказівник пам'яті слот 0, який у поточному контексті виклику є адресою контракту fibonacciLibrary, яка буде досить великою, якщо інтерпретувати її в тип даних uint. Таким чином, цілком імовірно, що виникне помилка та функція withdraw() виконає повернення до попереднього стану, оскільки вона не міститиме достатню кількість Ethereum, що в даному випадку дорівнює значенню uint(fibonacciLibrary), що повертає змінна calculatedFibNumber.

Ще гірше те, що смарт-контракт FibonacciBalance дозволяє користувачам викликати всі функції бібліотеки fibonacciLibrary через резервну функцію в рядку 26. Цей виклик може включати функцію setStart(). Ця функція дозволяє будь-кому змінювати або встановлювати слот пам'яті 0. У цьому випадку слот пам'яті 0 є адресою смарт-контракту fibonacciLibrary. Таким чином, зловмисник може створити шкідливий контракт, перетворити адресу в тип даних uint, а потім викликати функцію setStart та передати їй в якості параметру адресу шкідливого смарт-контракту. Це змінить fibonacciLibrary на адресу шкідливого контракту. Потім, щоразу, коли користувач викликає функцію withdraw() або резервну функцію, шкідливий контракт буде запущено, що може призвести до викрадення всього балансу смарт-контракту, оскільки була змінена фактична адреса для змінної fibonacciLibrary. Прикладом такого контракту про атаку може бути смарт-контракт, зображений на малюнку 2.16.

```
contract Attack {
    uint storageSlot0; // corresponds to fibonacciLibrary
    uint storageSlot1; // corresponds to calculatedFibNumber

    // fallback - this will run if a specified function is not found
    function() public {
        storageSlot1 = 0; // we set calculatedFibNumber to 0, so that if withdraw
        // is called we don't send out any ether.
        <attacker_address>.transfer(this.balance); // we take all the ether
    }
}
```

Рисунок 2.16 – Смарт-контракт для атаки зі зміною пам’яті.

Даний атакуючий контракт змінює calculatedFibNumber, змінюючи слот пам’яті 1. Зловмисник при цьому може змінити будь-які інші слоти пам’яті, які він вибере, щоб виконувати всі види атак на цей контракт.

2.13 Висновок

В даному розділі був проведений огляд типових вразливостей смарт-контрактів в блокчейні Ethereum. В даному списку приведені вразливості як програмного роду, які може допустити розробник при написанні смарт-контракту, так і ті вразливості, які зловмисник може використати при уже написаному контракті методом взаємодії з ним або написання шкідливих смарт-контрактів для цього. Хоча продемонстровані вразливості мають різну механіку реалізації всі вони можуть призвести фактично до втрати всіх активів, що пов’язані зі смарт-контрактом, на який була проведена атака. З розвитком технологій блокчейну, розвивається і індустрія шахраїв. В майбутньому можуть виникнути нові вразливості, але на даний момент для того, щоб вважати смарт-контракт захищеним, необхідно провести заходи захисту проти наведених в даному розділі вразливостей.

РОЗДІЛ 3. СПЕЦІАЛЬНА ЧАСТИНА

На даний момент технології блокчейну використовуються у багатьох галузях. Найбільш часто смарт-контракти використовуються для створення токенів, які служать внутрішньою валютою для сервісу, який представляє його власник. Наразі розробники смарт-контрактів активно стараються розробляти безпечні контракти, тому що навіть невелика похибка або помилка в коді можуть призвести до непоправних наслідків. Програмний код смарт-контракту не можна змінити після його деплою в блокчейн. Таким чином про захист від вразливостей треба піклуватись на етапі розробки смарт-контракту. Частіше за все розробники самі стараються захистити смарт-контракт від можливих атак шляхом реалізації захисних механізмів в програмному коді смарт-контракту. Але існують і спеціальні команди, які проводять аудит смарт-контрактів та вказують на зроблені помилки і можливі варіанти успішних атак на них, а також надають рекомендації по усуненню даних вразливостей.

3.1 Проведення аудиту смарт-контракту PPSD.sol

Аудит смарт-контрактів відіграє велику роль в процедурі їх захисту від вразливостей. Часто токени, зроблені за допомогою смарт-контрактів виступають внутрішньою валютою для якогось крупного сервісу. При розробці даного сервісу компанія, яка цим займається, залучає фінансування з боку інвесторів за допомогою процедури під назвою ICO. Дана процедура являє собою процес залучення інвестицій в проект шляхом створення пакетів токенів, які інвестори можуть отримати взамін на вкладені гроші. Для цього розробниками створюється спеціальний документ під назвою whitepaper, який презентується на розгляд інвесторам та в якому описують цілі проекту, його етапи, які проблеми він вирішує, бізнес-модель та інше. Тому професійний аудит може зберегти не тільки токени, які випускаються, а і гроші інвесторів, які були вкладені в даний проект. Далі приводяться вразливості, які були виявлені при проведенні аудиту смарт-контракту PPSD.sol.

- В даному смарт-контракті присутні функція changePrice, яка дозволяє змінювати ціну токенів при їх покупці користувачами, функція changeForwardingAddress, яка змінює адресу для пересилки криптовалюти при покупці токенів та функція changeOwner, яка дозволяє змінювати власника смарт-контракту. Дані функції змінюють стан смарт-контракту тому являються найбільш важливими з точки зору захисту від атак. В даному випадку ці функції є відкритими для всіх користувачів, відповідно зловмисник може скористатись відсутність контролю доступу та змінити дані смарт-контракту так, як він захоче, наприклад, змінити адресу власника на свою власну або встановити ціну токенів на дуже маленьке значення та закупити їх по новій ціні.
- В функції receive при пересилці криптовалюти на вказану адресу гаманця використовується функція send, але відсутня перевірка значення, яке вона повертає після виконання. Це може призвести до того, що зловмисник може відправити транзакцію з помилкою (наприклад, мінімальну кількість криптовалюти Ethereum, яка буде недостатньою для покупки токенів), але через відсутність перевірки результуючого значення даної функції, токени все одно будуть відправлені на адресу зловмисника.
- В функції retrieveTokens даного смарт-контракту виконується перевірка на те, що користувач, який викликає дану функцію, є власником смарт-контракту. Це відбувається за допомогою порівняння змінних tx.origin та owner. Зловмисник може скористатись цим, створивши новий смарт-контракт, який на перший погляд не буде викликати підозри, та переконати власника смарт-контракту PPSD.sol виконати якусь функцію контракту-шахрая. Такий контракт в свою чергу викличе функцію retrieveTokens від імені власника даного контракту, що призведе до того, що дана перевірка буде пройдена і зловмисник зможе отримати на свій гаманець всі токени, які знаходяться на балансі смарт-контракту, як сказано в пункті 2.11 даної роботи.

- Функція receive, яка приймає криптовалюту від користувачів та взамін відправляє токени, має в собі рядок:

  uint40 amount = msg.value * (10 ** decimals()) / price;

  Даний рядок коду виконує розрахунок кількості токенів, які необхідно відправити користувачеві залежно від кількості криптовалюти, яку він прислав на смарт-контракт. В Solidity тип даних uint40 може зберігати значення від 0 до 2^40 - 1, тому якщо користувач відправить на смарт-контракт велику суму криптовалюти, ця сума після множення на 10^decimals() може стати занадто великою та відбудеться переповнення змінної amount. Таким чином користувач при достатньо великій інвестиції отримає малу кількість токенів на свій криптогаманець.
- Функції changePrice, changeForwardingAddress та changeOwner потребують не тільки введення контролю доступу, а і перевірки значень, які передаються в них як параметри. Навіть якщо доступ до них буде мати тільки власник, він може помилитись і вказати неправильне значення, яке він хоче змінити. Якщо, наприклад, в функціях changePrice та changeForwardingAddress у власника буде можливість заново викликати дані функції та виправити неправильно вказані значення (при умові, якщо він зверне на це увагу та зрозуміє, що допустив помилку), то в випадку з функцією changeOwner неправильно вказана адреса нового власника в параметрі функції може призвести до повної втрати доступу до виклику функцій смарт-контракту, які потребують привілеїв власника.

В таблиці 3.1 була зібрана коротка інформація про виявлені в процесі аудиту вразливості. Кожній з них було надано код для спрощення подальших звернень до них.

Таблиця 3.1 – Список вразливостей смарт-контракту PPSD.sol

Опис вразливості	Тип вразливості	Код вразливості
Доступ будь-якого користувача до функцій, які змінюють стан смарт-контракту	Контроль доступу	В1
Відсутність перевірки результуючого значення при виклику функції send в функції receive	Відсутність перевірки значень, що повертаються функціями низького рівня	В2
Перевірка адреси, яка викликає функцію retrieveTokens за допомогою tx.origin	Автентифікація за допомогою tx.origin	В3
Відсутність перевірки змінної amount в функції receive на переповнення при розрахунку кількості токенів, які необхідно відправити користувачеві	Арифметичні проблеми	В4
Відсутність перевірки значень, які передаються в параметри функцій, які змінюють стан контракту	Необроблені винятки	В5

3.2 Реалізація заходів щодо захисту смарт-контракту PPSD.sol від виявлених вразливостей

При проведенні аудиту смарт-контракту було виявлено 5 критичних вразливостей, які можуть призвести до таких непоправних наслідків як втрата токенів, які зберігаються на даному контракті або втрата доступу до основних функцій смарт-контракту. Так як всі дії в блокчейні є невідворотними, дані вразливості повинні бути виправлені до того, як буде виконаний процес деплою смарт-контракту в основну мережу блокчейну Ethereum – Mainnet.

Вразливість В1

Контроль доступу в смарт-контрактах реалізується за допомогою введення змінної owner, яка відповідає адресі власника контракту та присвоєнні цій

змінній адреси, яка виконує деплой смарт-контракту в блокчейн. Дана адреса виконує роль адміністратора та має більш розширені повноваження ніж інші користувачі. Далі, за рекомендаціями експертів, в функціях, доступ до яких необхідно обмежити для всіх окрім власника необхідно проводити перевірку на те, що адреса, яка викликає функцію, співпадає з адресою власника даного смарт-контракту. Це можна виконати за допомогою функції мови Solidity – require. Дана функція приймає 2 параметри. Першим параметром виступає умова, яка перевіряється, результатом якої повинно бути булеве значення true/false. Другим параметром є попереджувальне повідомлення, яке буде бачити користувач, якщо дана перевірка не пройде. В даному випадку для контролю доступу до функцій смарт-контракту, які змінюють стан контракту, може використовуватись перевірка, яка має наступний вигляд:

require(tx.origin == owner, "This operation is only allowed to the owner.");

В початковому коді смарт-контракту PPSD.sol така перевірка є тільки в функції retrieveTokens, відповідно її треба додати також в функції changePrice, changeForwardingAddress та changeOwner. На рисунку 3.1 показано як будуть виглядати дані функції після додавання даних перевірок.

```
function changePrice(uint256 newPrice) public {
    require(tx.origin == owner, "This operation is only allowed to the owner.");
    price = newPrice;
}

function changeForwardingAddress(address newAddress) public {
    require(tx.origin == owner, "This operation is only allowed to the owner.");
    forwardingAddress = payable(newAddress);
}

function changeOwner(address newOwner) public {
    require(tx.origin == owner, "This operation is only allowed to the owner.");
    owner = newOwner;
}
```

Рисунок 3.1 – Стандартна реалізація контролю доступу для смарт-контракту PPSD.sol.

Але при великому обсязі смарт-контракту та великій кількості таких функцій, дане рішення може нагромаджувати і без того велику кількість коду смарт-контракту. Також при деплої смарт-контракту в блокчейн, сума комісії прямо пропорційно залежить від кількості символів переведеного в байткод вихідного коду контракту. Тому для оптимізації даного рішення пропонується розробити модифікатор доступу для перевірки того, що ініціатор транзакції являється власником смарт-контракту. Модифікатори функцій можна використовувати для декларативного внесення змін до семантики функцій.[16] Даний модифікатор приведений на рисунку 3.2.

```
modifier onlyOwner {
    require(msg.sender == owner, "This operation is only allowed to the owner.");
    _;
}
```

Рисунок 3.2 – Модифікатор доступу onlyOwner.

Символ «_;» називається підстановковим знаком. Він поєднує код функції із кодом модифікатора. Модифікатор функції обо'вязково повинен містити в собі даний символ. Іншими словами, тіло функції, до якої приєднаний модифікатор, буде вставлено туди, де у визначенні модифікатора з'являється даний символ. Відповідно, в процесі виконання функції, до якої буде приєднаний даний модифікатор спочатку буде виконувати перевірку, а потім тіло самої функції.

На рисунку 3.3 демонструється вигляд функцій, які потребують контролю доступу після введення запропонованого рішення.

```
modifier onlyOwner {
    require(msg.sender == owner, "This operation is only allowed to the owner.");
    _;
}

function changePrice(uint256 newPrice) public onlyOwner {
    price = newPrice;
}

function changeForwardingAddress(address newAddress) public onlyOwner {
    forwardingAddress = payable(newAddress);
}

function changeOwner(address newOwner) public onlyOwner {
    owner = newOwner;
}
```

Рисунок 3.3 – Реалізація модифікатора функцій для контролю доступу.

Вразливість В2

В функції receive початкового смарт-контракту, наведеного в Додатку В, для відправки криптовалюти на зазначену адресу використовується функція send. Часто розробникам рекомендують в якості вирішення даної вразливості перевіряти значення, яке повертає функція, та поміщати цю перевірку в функцію require. Це зробить подальше виконання тіла функції неможливим, якщо транзакція буде помилковою та функція send поверне значення false. Дане рішення показано на рисунку 3.4.

```
receive() external payable {
    uint40 amount = msg.value * (10 ** decimals()) / price;
    _transfer(address(this), tx.origin, amount);
    require(forwardingAddress.send(address(this).balance), "Transaction error");
}
```

Рисунок 3.4 – Перевірка значення, яке повертає функція send.

Натомість, пропонується замість функції send використовувати функцію transfer, яка у випадку неуспішної транзакції викликає функцію revert. Дана функція зупиняє виконання смарт-контракту, скасовує всі зміни, внесені під час виконання, і повертає залишок газу абоненту без використання додаткових перевірок. Використання функції transfer являється більш універсальним та

сучасним методом для відправки криптовалюти, так як вона не потребує додаткових перевірок та в разі виникнення помилки в транзакції, відмінить її автоматично. На рисунку 3.5 приведено функцію receive після реалізації даного рішення в смарт-контракті.

```
receive() external payable {
    uint40 amount = msg.value * (10 ** decimals()) / price;
    _transfer(address(this), tx.origin, amount);
    forwardingAddress.transfer(address(this).balance);
}
```

Рисунок 3.5 – Заміна функції send на transfer.

Вразливість В3

Дана вразливість небезпечна тим, що при використанні функції tx.origin для автентифікації користувача, ця функція в будь-якому випадку повертає адресу початкового ініціатора транзакції або ланцюга транзакцій. Як було сказано в пункті 2.11 даної роботи, зловмисник може використати такий тип автентифікації на свою користь та викликати функції смарт-контракту, на який він проводить атаку від імені його власника. Наприклад, є ланцюжок викликів Функція а ← Контракт А ← Функція б ← Контракт Б. Контрактом А виступає смарт-контракт, Функція а якого автентифікує користувача за допомогою tx.origin. Контракт Б – в даному випадку являється зловмисним контрактом, який має в собі Функцію б, яка звертається до Контракту А та викликає Функцію а. Якщо автор зловмисного контракту обманом змусить власника Контракту А викликати Функцію б (відповідно, ставши її ініціатором), то пройшовши через весь ланцюг викликів, Функція а виконається від імені власника Контракту А. Це відбувається тому, що функція tx.origin не може повертати адресу смарт-контракту. Вона завжди повертає адресу користувача, який стоїть на початку ланцюга. Тому, враховуючи даний недолік функції tx.origin, для вирішення даної проблеми рекомендується замінити її на функцію msg.sender, яка, в свою чергу, може дорівнювати адресі смарт-контракту та при виклику повертає тільки ту адресу, яка стоїть на одну сходинку вище в даному ланцюжку викликів.

Відповідно, в випадку з Контрактом А та Контрактом Б якщо власник Контракту А виконає ті ж самі дії, то при перевірці адреси ініціатора функція msg.sender поверне адресу Контракту Б, яка не являється адресою власника Контракту А. В такому разі зловмисник не зможе виконати дії, які запланував. Таким чином функції смарт-контракту PPSD.sol, які використовували tx.origin тепер будуть мати вигляд, як показано на рисунку 3.6.

```
constructor() ERC20("PPSD", "PPSD") {
    owner = msg.sender;
    _mint(owner, 1_000_000 * (10 ** uint256(decimals())));
    _mint(address(this), 1_000_000 * (10 ** uint256(decimals())));
}

modifier onlyOwner {
    require(msg.sender == owner, "This operation is only allowed to the owner.");
    _;
}

receive() external payable {
    uint256 amount = msg.value * (10 ** decimals()) / price;
    _transfer(address(this), msg.sender, amount);
    forwardingAddress.transfer(address(this).balance);
}
```

Рисунок 3.6 – Заміна функції tx.origin на msg.sender.

Вразливість В4

Арифметичні проблеми, такі як переповнення змінних, присутні майже у всіх відомих мовах програмування. Solidity, як і весь блокчейн, не підтримує числа з плаваючою комою. Дана мова програмування працює тільки з цілими числами. Криптовалюта, в свою чергу, може мати дрібне значення. Наприклад, на момент написання даної роботи 100$ дорівнює 0.002 Біткойна. Тому, для коректного відображення кількості токенів в гаманці користувача, смарт-контракти використовують змінну decimals. Дана змінна відображає максимальну кількість цифр після коми, яку може мати токен. Наприклад, якщо змінна decimals буде дорівнювати 1, то 1 токен можна буде розділити тільки на 10 частин (1 знак після коми). Для запобігання неправильних розрахунків при передачі токенів, стандартом прийнято встановлювати в змінну decimals

значення 18. Тобто якщо користувач має в своєму гаманці 1 токен, то всередині смарт-контракту його баланс відображається як 1 * 10^18. В функції receive смарт-контракту PPSD.sol присутній наступний рядок коду:

```
uint40 amount = msg.value * (10 ** decimals()) / price;
```

В змінній msg.value зберігається кількість криптовалюти, яку користувач прислав на смарт-контракт. Ціна в даному випадку дорівнює 37 * 10^13. Слід зазначити, що криптовалюта Ethereum також має 18 знаків після коми. Наприклад, користувач відправить на даний смарт-контракт 1 Ether. В такому разі дану формулу розрахунку можна переписати в такому вигляді:

```
uint40 amount = 1 * 10^18 * 10^18 / (37 * 10^13);
```

Згідно пріоритету виконання математичних операцій в мові програмування Solidity[17], зведення в ступінь буде виконуватись в першу чергу. За ним ідуть операції множення та ділення, які мають однаковий пріоритет, а значить будуть виконуватись послідовно зліва направо. Таким чином, спочатку буде виконана операція множення, яка в результаті дасть 10^36. Далі це значення ділиться на ціну токену та отримується кількість токенів, яку необхідно відправити користувачеві. Але в випадку якщо користувач відправить на даний смарт-контракт більше ніж 99999 Ether, то відбудеться переповнення змінної amount при виконанні множення. Наприклад, користувач відправить 100000 Ether на даний контракт. Приведений рядок коду з розрахунками буде виглядати наступним чином:

```
uint40 amount = 10^5 * 10^18 * 10^18 / (37 * 10^13);
```

Після перемноження перших 3 значень формули повинен вийти результат, який дорівнює 10^41, але так як тип даних uint40 може приймати максимальне значення 10^40 – 1, в цей момент відбудеться переповнення змінної amount і вона прийме значення 10. Далі це значення ділиться на ціну токену, а так як вона є на багато порядків вище за дане значення, результатом розрахунку кількості токенів, які необхідно відправити користувачеві, буде 0. Таким чином при

значній інвестиції користувач не отримає нічого. Зазвичай рекомендацією для рішення даної вразливості є використання бібліотеки SafeMath, яка розроблена командою OpenZeppelin. Дана бібліотека враховує можливі випадки переповнення та викликає помилку, якщо таке відбулось. Лістинг бібліотеки SafeMath.sol наведено в Додатку Ґ.

Натомість для вирішення арифметичного переповнення при розробці смарт-контрактів рекомендується використовувати компілятор Solidity версії 0.8.0 і вище. В дану версію компілятора було додано перевірки на переповнення за замовчуванням. Таким чином запропоноване рішення дозволить позбавитись від даної бібліотеки і тим самим зменшити кількість байткоду смарт-контракту та знизити вартість його деплою в блокчейн. Також для запобігання переповнень необхідно змінити тип даних змінної amount з uint40 на uint256, яка може зберігати в собі значення від 0 до 10^256 – 1. Дана зміна суттєво знизить вірогідність виникнення переповнення при виконанні арифметичних операцій за участю великих чисел. На рисунку 3.7 показана функція receive після реалізації даного рішення.

```
receive() external payable {
    uint256 amount = msg.value * (10 ** decimals()) / price;
    _transfer(address(this), msg.sender, amount);
    forwardingAddress.transfer(address(this).balance);
}
```

Рисунок 3.7 – Зміна типу даних змінної для уникнення переповнення.

Вразливість B5

Функції, які приймають параметри завжди повинні перевіряти валідність даних параметрів. Навіть якщо доступ до таких функцій буде тільки у власника смарт-контракту, не можна виключати людський фактор, який може спричинити помилку з боку власника. Дана помилка може призвести до фатальних наслідків, в тому числі і до повної втрати доступу до смарт-контракту з боку його власника.

Дану проблему пропонується вирішити введенням перевірки на валідність аргументів функцій, які їх приймають. В функцію changePrice, яка змінює ціну, буде додано перевірку на те, що нова ціна, яка вказується в параметрі, не дорівнює 0. Така перевірка не дозволить встановити ціну токену в 0 навіть помилково. Функція require, яка використовується для виконання перевірок, відмінить транзакцію в разі спроби передання 0 в параметри даної функції та поверне стан смарт-контракту в момент, коли транзакція не була проведена. Дана перевірка буде мати наступний вигляд:

```
require(newPrice != 0, "Token price cannot be 0.");
```

В випадку виникнення помилки, повідомлення "Token price cannot be 0." буде показано ініціатору транзакції для розуміння причини відміни такої транзакції.

Функції ChangeOwner та changeForwardingAddress в якості параметру приймають адресу для пересилки криптовалюти Ethereum та адресу нового власника смарт-контракту відповідно. Для даних функцій також рекомендується ввести перевірку на те, що нова адреса не може дорівнювати 0. Адреса в блокчейні – це число в форматі HEX (шістнадцяткова система числення), тому при вказанні 0 в якості параметра, який повинен містити в собі адресу, таке значення не буде вважатись помилковим і, відповідно, транзакція не буде відмінена. Така перевірка виглядатиме так:

```
require(newAddress != address(0), "New address can't be null.");
```

Це рішення автоматично буде відхиляти спроби встановлення змінної адреси на 0, таким чином навіть при виникненні помилки з боку власника смарт-контракту (або цілеспрямованих дій), система буде відміняти дані спроби змінити адресу.

Адреси в блокчейні Ethereum, як було описано раніше, виступають як шістнадцяткове значення, які завжди мають однакову довжину – 40 байт. Мова програмування Solidity не має достатнього функціоналу для того, щоб перевіряти

вміст змінної, що відповідає адресі, але при цьому дозволяє перевіряти довжину значення, яке передається в параметри функції. Дану перевірку пропонується реалізувати за допомогою окремої функції, яка буде повертати значення true або false. Реалізація даної функції показана на рисунку 3.8.

```
function isValidAddress(address addr) internal view returns (bool) {
    uint256 size;
    assembly { size := extcodesize(addr) }
    return (size == 40);
}
```

Рисунок 3.8 – Функція для перевірки довжини адреси.

В даному випадку функція extcodesize дозволяє дізнатись довжину змінної addr в байтах. Після цього функція перевірки порівнює отримане значення з числом 40 та повертає true, якщо довжина адреси правильна або false в іншому випадку.

Сама ж перевірка реалізовується шляхом виклику даної функції перевірки та поміщення цього виклику в функцію require. Таким чином якщо в якості параметра функції, яка очікує нову адресу, буде передана закоротка або задовга адреса, дана перевірка не пройде і транзакція буде відхилена. Таке рішення дозволяє знизити ризик того, що в параметр функцій, які приймають адресу, буде передано помилкове значення нової адреси. Запропоновані заходи для вирішення даної вразливості демонструються на рисунку 3.9.

```
function isValidAddress(address addr) internal view returns (bool) {
    uint256 size;
    assembly { size := extcodesize(addr) }
    return (size == 40);
}

function changePrice(uint256 newPrice) public onlyOwner {
    require(newPrice != 0, "Token price cannot be 0.");
    price = newPrice;
}

function changeForwardingAddress(address newAddress) public onlyOwner {
    require(newAddress != address(0), "New address can't be null.");
    require(isValidAddress(newAddress), "Invalid address specified");
    forwardingAddress = payable(newAddress);
}

function changeOwner(address newOwner) public onlyOwner {
    require(newOwner != address(0), "New address can't be null.");
    require(isValidAddress(newOwner), "Invalid address specified.");
    owner = newOwner;
}
```

Рисунок 3.9 – Введення в смарт-контракт перевірок параметрів функцій.

3.3 Висновок

В даному розділі було проведено аудит смарт-контракту PPSD.sol, за допомогою якого було виявлено найбільш критичні для даного контракту вразливості. Далі було запропоновано та реалізовано заходи щодо захисту смарт-контракту від виявлених вразливостей. За допомогою реалізованих програмних методів захисту, смарт-контракт PPSD.sol був підготовлений до подальшого деплою в основну мережу блокчейну Ethereum – Mainnet.

РОЗДІЛ 4. ЕКОНОМІЧНИЙ РОЗДІЛ

4.1 Постановка задачі

У даному розділі кваліфікаційної роботи наводиться техніко-економічне обґрунтування(ТЕО) заходів захисту смарт-контракту PPSD.sol, які були наведені у спеціальному розділі.

Техніко-економічне обґрунтування (ТЕО) – це обов'язкова складова частина будь-якого інвестиційного проекту, тобто проекту, що потребує певних фінансових витрат. Основна мета розробки ТЕО – дати фінансову оцінку передбачуваних витрат та одержуваного корисного результату, а також оцінити прибутковість проекту і, в кінцевому підсумку, економічну доцільність його розробки та впровадження.

Для успішного проведення техніко-технічного обґрунтування необхідно виконати наступні дії:

– Виконати розрахунок капітальних витрат на придбання і налагодження програмного і апаратного забезпечення для реалізації методів, які наведені у спеціальному розділі;

– Розрахувати річні експлуатаційні витрати на утримання і обслуговування програмного та апаратного забезпечення;

– Визначити річний економічний ефект від реалізації методів захисту;

– Визначити та провести аналіз показників економічної ефективності запропонованих у спеціальному розділі методів;

– Сформулювати висновок щодо економічної доцільності обраних методів захисту смарт-контракту.

4.2 Виконання розрахунків

4.2.1 Розрахунок капітальних витрат

Капітальні інвестиції – це кошти, призначені для створення і придбання основних фондів і нематеріальних активів, що підлягають амортизації.

До капітальних витрат відносяться витрати на впровадження методів захисту представленого смарт-контракту, які визначаються виходячи з трудомісткості впровадження цих методів.

Трудомісткість реалізації запропонованих методів визначається тривалістю кожної робочої операції, починаючи з складання технічного завдання і закінчуючи оформленням документації (за умови роботи одного спеціаліста з інформаційної безпеки).

Трудомісткість визначається за наступною формулою:

$$t = tтз + tв + tа + tозб + tр + tд \quad (4.1)$$

де $tтз$ – тривалість складання технічного завдання на впровадження методів;

$tв$ – тривалість вивчення ТЗ, літературних джерел за темою тощо;

$tа$ – тривалість проведення аудиту смарт-контракту;

$tозб$ – тривалість вибору основних рішень з забезпечення безпеки смарт-контракту;

$tр$ – тривалість програмної реалізації методів захисту смарт-контракту;

$tд$ – тривалість документального оформлення методів.

Визначено, що, враховуючи особливості запропонованих методів, наведені вище величини становлять: $tтз$ = 24 години, $tв$ = 36 годин, $tа$ = 24 години, $tозб$ = 20 годин, $tр$ = 24 години, $tд$ = 8 годин.

Відповідно,

$$t = 24 + 36 + 24 + 20 + 24 + 8 = 136 \text{ годин}$$

Розрахунок витрат на впровадження методів захисту смарт-контракту

Вартість 1 години машинного часу ПК визначається за формулою:

$$Смч = P \cdot tнал \cdot Ce + \frac{Фзал \cdot На}{Fp} + \frac{Клпз \cdot Напз}{Fp} = 0,8 \cdot 5 \cdot 1,68 = 6,72\, грн, \qquad (4.2)$$

де P – встановлена потужність ПК, кВт;

Ce – тариф на електричну енергію, грн/кВт*година;

$Фзал$ – залишкова вартість ПК на поточний рік, грн;

$На$ – річна норма амортизації на ПК, частки одиниці;

$Напз$ – річна норма амортизації на ліцензійне програмне забезпечення, грн;

Fp – річний фонд робочого часу (за 40-годинного робочого тижня Fp = 1920).

Вартість машинного часу для розробки методів захисту смарт-контракту визначається за формулою:

$$Змч = t \cdot Смч = 136 \cdot 6,72 = 913,92\, грн, \qquad (4.3)$$

де tp – тривалість розробки методів захисту, годин;

$tд$ – тривалість документального оформлення методів, годин;

$Смч$ – вартість 1 години машинного часу ПК, грн/кВт*година.

Заробітна плата виконавця враховує основну і додаткову заробітну плату, а також відрахування на соціальні потреби (пенсійне страхування, страхування на випадок безробіття, тощо) і визначається за формулою:

$$Ззп = t \cdot Зпр = 136 \cdot 37,125 = 5049\, грн, \qquad (4.4)$$

де t – загальна тривалість розробки методів захисту смарт-контракту, годин;

$Зпр$ – середньогодинна заробітна плата блокчейн розробника з нарахуваннями, грн/година.

Витрати на впровадження запропонованих методів захисту $Кпз$ складаються з витрат на заробітну плату виконавця програмного забезпечення $Ззп$ і вартості витрат машинного часу, що необхідний для цього $Змч$:

$$Кпз = Ззп + Змч = 5049 + 913,92 = 5962,92 \qquad (4.5)$$

Відповідно до запропонованих методів захисту смарт-контракту PPSD.sol, було виконано аудит смарт-контракту та було заплановано деплой смарт-контракту PPSD.sol в блокчейн Ethereum. Витрати на дані процедури:

- аудит смарт-контракту – 500$ ≈ 13500 грн;
- деплой смарт-контракту в блокчейн – 455$ ≈ 12285 грн.

Таким чином, капітальні витрати на впровадження методів підвищення рівня інформаційної безпеки дорівнюють:

$$K = K\text{пз} + K\text{а} + K\text{д} = 5962{,}92 + 13500 + 12285 = 31748\text{грн} \quad (4.6)$$

де $K\text{пз}$ – витрати на впровадження запропонованих методів захисту;

$K\text{а}$ – витрати на аудит смарт-контракту;

$K\text{д}$ – витрати на деплой смарт-контракту в блокчейн.

4.2.2 Розрахунок річних експлуатаційних витрат

Експлуатаційні витрати – це поточні витрати на експлуатацію та обслуговування об'єкта проектування за календарний рік, визначений у грошовій формі.

До поточних витрат відносяться наступні витрати:

- вартість Upgrade-відновлення й модернізації системи($C\text{в}$);
- витрати на керування системою($C\text{к}$);
- витрати, викликані активністю користувачів системи($C\text{ак}$).

Під «витратами на керуванням системою» маються на увазі витрати, пов'язані з керуванням і адмініструванням серверів та інших компонентів системи. До цієї статті витрат відносяться наступні витрати:

- навчання адміністративного персоналу й кінцевих користувачів;
- амортизаційні відрахування від вартості обладнання та ПЗ;
- заробітна плата обслуговуючого персоналу;
- аутсорсинг;

- навчальні курси та сертифікація обслуговуючого персоналу;
- технічне й організаційне адміністрування й сервіс

Так як програмний код, який буде знаходитись в блокчейні після деплою, не може бути змінений, витрати на адміністрування зводяться до витрат на підтримку ліквідності токену PPSD з боку замовника($Cл$). Витрати на підтримку ліквідності токену – це витрати, які зумовлюють періодичну покупку токенів самим замовником для підтримання його курсу. Якщо власники даних токенів будуть тільки продавати їх, то курс такої валюти не буде рости через те, що пропозиція буде значно перевищувати попит. Таким чином було заплановано щомісячні витрати на підтримання ліквідності токену в розмірі 500$ (13500 грн). Також за процедуру купівлі токенів в блокчейні Ethereum буде враховуватись комісія($Cк$) в районі 40$ (1080 грн). Відповідно, витрати на керування даним проектом дорівнюють:

$$Cк = (Cл + Cком) \cdot 12 = (13500 + 1080) \cdot 12 = 174960 грн \tag{4.7}$$

Таким чином, річні поточні витрати на функціонування розробленого токену та підтримки його курсу складають 174960 грн.

4.2.3 Визначення річного економічного ефекту

Кінцевим результатом впровадження заходів щодо забезпечення безпеки інформації є величина відвернених втрат, що розраховується виходячи з імовірності виникнення інциденту інформаційної безпеки й можливих економічних втрат від нього.

В даному випадку через те, що програмний код в блокчейні неможливо змінити, в разі успішної атаки на смарт-контракт з боку зловмисника, власник смарт-контракту ризикує втратити всі токени, які знаходяться на балансі смарт-контракту або доступ до них. Таким чином, величина можливих збитків визначається за формулою:

$$B = Q \cdot P, грн, \tag{4.8}$$

де Q – кількість токенів на балансі смарт-контракту;

P – ціна за один токен, грн.

Заплановано, що вартість 1 токену буде сягати 27 грн. За допомогою смарт-контракту PPSD.sol буде випущено 2000000 токенів, 1000000 з яких буде записана на баланс смарт-контракту для подальшої можливості їх купівлі користувачами.

Відповідно, величина можливих збитків від реалізації загроз буде складати:

$$B = 1000000 \cdot 27 = 27000000 \text{ грн}$$

Загальний ефект від впровадження методів захисту смарт-контракту від виявлених вразливостей з урахуванням вірогідності реалізації даних вразливостей:

$$E = B \cdot R - C \text{, грн} \qquad (4.9)$$

де B – загальний збиток від реалізації атаки на смарт-контракт, *грн*;

R – вірогідність успішної реалізації атаки на смарт-контракт, частки одиниці (R = 40%);

C – щорічні витрати на експлуатацію проекту з продажу токенів, *грн.*

Загальний ефект від реалізації методів захисту смарт-контракту визначається з урахуванням вірогідності проведення атаки на нього:

$$E = 27000000 \cdot 0,4 - 174960 = 10625040 \text{грн}$$

4.2.4 Визначення та аналіз показників економічної ефективності

Коефіцієнт повернення інвестицій *ROSI* показує, скільки гривень додаткового прибутку приносить одна гривня капітальних інвестицій на впровадження системи інформаційної безпеки:

$$ROSI = \frac{E}{K}, \text{ частки одиниці} \quad (4.10)$$

де E – загальний ефект від реалізації захисних методів, грн;

K – капітальні інвестиції за варіантами, що забезпечили цей ефект, грн.

Коефіцієнт повернення інвестицій $ROSI$ дорівнює:

$$ROSI = \frac{10625040}{31748} = 334{,}66, \text{ частки одиниці}$$

Проект визнається економічно доцільним, якщо розрахункове значення коефіцієнта повернення інвестицій перевищує величину річної депозитної ставки з урахуванням інфляції:

$$ROSI > (N\text{деп} - N\text{інф}) / 100 \quad (4.11)$$

де $N\text{деп}$ – річна депозитна ставка, (11%);

$N\text{інф}$ – річний рівень інфляції, (10%).

Розрахункове значення коефіцієнта повернення інвестицій:

$$334{,}66 > (11 - 10) / 100 = 334{,}66 > 0{,}01$$

Термін окупності капітальних інвестицій To показує, за скільки років капітальні інвестиції окупляться за рахунок загального ефекту від реалізації захисних мір для смарт-контракту:

$$To = \frac{K}{E} = \frac{1}{ROSI} = \frac{1}{334{,}66} = 0{,}003 \text{ роки} \quad (4.12)$$

4.3 Висновок

При проведенні техніко-технічного обґрунтування було визначено, що розмір капітальних витрат на розробку захисних мір для смарт-контракту PPSD.sol складає 31748 грн, річні поточні витрати на функціонування випущених токенів складають 174960 грн.

Розробка методів захисту смарт-контракту PPSD.sol від виявлених при проведенні аудиту вразливостей є економічно доцільним, оскільки коефіцієнт повернення інвестицій ROSI складає 334,66 грн/грн, що означає отримання 334,66 грн економічного ефекту на кожну гривню капітальних вкладень на впровадження методів захисту смарт-контракту. Отримане значення коефіцієнту повернення інвестицій значно вище дохідності альтернативного вкладення коштів. Термін окупності при цьому складатиме 0,003 роки (приблизно 1,095 днів).

Розрахунки в даному розділі кваліфікаційної роботи були виконані відповідно до методичних вказівок [19].

ВИСНОВКИ

На даний момент технології блокчейну набули високої популярності за рахунок децентралізації, прозорості, автоматизованості, відсутності посередників при проведенні транзакцій та ін. Автоматизація виконання транзакцій в блокчейні досягається за рахунок розробки та деплою в мережу смарт-контрактів. Проблема полягає в тому, що всі дії в блокчейні являються невідворотними. Саме тому смарт-контракт, який потрапив в блокчейн, вже не може бути змінений. Відповідно, безпеці смарт-контрактів треба приділяти особливе значення в процесі їх розробки.

В даній кваліфікаційній роботі виконано реалізацію методів захисту смарт-контрактів від вразливостей, які можуть виникнути при їх розробці та функціюванні.

У першому розділі роботи була надана інформація про блокчейн, принципи його роботи, переваги порівняно з нинішньою банківською системою. Також було дано визначення смарт-контрактів, описано процес їх розробки та необхідне для цього програмне забезпечення. Далі було проведено опис демонстраційного смарт-контракту PPSD.sol, наведений список функцій смарт-контракту та надано пояснення їх функціоналу.

В другому розділі було продемонстровано вразливості смарт-контрактів, які найчастіше зустрічаються в блокчейні. Для кожної вразливості було надано пояснення механізмів її реалізації та проілюстровані приклади злоякісних смарт-контрактів для проведення атаки, яка реалізує дану вразливість.

В спеціальній частині було проведено аудит демонстраційного смарт-контракту PPSD.sol та визначені п’ять основних вразливостей, які присутні в даному контракті. Далі було надано опис кожної вразливості та описано механізми їх реалізації. Слідом було запропоновано та реалізовано методи щодо захисту смарт-контракту від виявлених вразливостей та надано порівняння ефективності даних методів з методами, які зазвичай застосовуються розробниками смарт-контрактів.

В четвертому розділі роботи приведено обґрунтування економічної доцільності обраних методів захисту смарт-контракту PPSD.sol. Результати виконання розрахунків показали, що обрані методи є економічно доцільними, позаяк коефіцієнт повернення інвестицій ROSI складає 334,66 грн/грн. Термін окупності витрат на запропоновані методи становить 1 день.

Таким чином, враховуючи вище описане, можна стверджувати, що при виконанні кваліфікаційної роботи були досягнуті поставлені цілі щодо захисту смарт-контракту PPSD.sol від вразливостей, які були виявлені в процесі аудиту.

Досягнутий в результаті проведеної роботи рівень безпеки дозволяє виконати деплой даного смарт-контракту в основну мережу блокчейну Ethereum – Mainnet та запускати продаж токенів в мережі. Дана кваліфікаційна робота була виконана відповідно до методичних вказівок [20].

ПЕРЕЛІК ПОСИЛАНЬ

1. Hayes A. Blockchain Explained [Електронний ресурс] / A. Hayes, J. Mansa, S. Kvilhaug. – 2022. – Режим доступу до ресурсу: https://www.investopedia.com/terms/b/blockchain.asp.

2. What Is a Smart Contract? [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://chain.link/education/smart-contracts.

3. Smart Contracts [Електронний ресурс] – Режим доступу до ресурсу: https://corporatefinanceinstitute.com/resources/knowledge/deals/smart-contracts/.

4. Non-fungible tokens (NFT) [Електронний ресурс] – Режим доступу до ресурсу: https://ethereum.org/en/nft/.

5. Ebun-Amu C. What Is Solidity and How Is It Used to Develop Smart Contracts? [Електронний ресурс] / Calvin Ebun-Amu. – 2021. – Режим доступу до ресурсу: https://www.makeuseof.com/what-is-solidity/.

6. Known Attacks [Електронний ресурс] // Ethereum Smart Contract Best Practices – Режим доступу до ресурсу: https://consensys.github.io/smart-contract-best-practices/known_attacks/.

7. Code With No Effects [Електронний ресурс] – Режим доступу до ресурсу: https://swcregistry.io/docs/SWC-135.

8. OWASP Top Ten [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-project-top-ten/.

9. Access Control [Електронний ресурс] // NCC Group – Режим доступу до ресурсу: https://dasp.co/#item-2.

10. Unchecked Return Values For Low Level Calls [Електронний ресурс] // NCC Group – Режим доступу до ресурсу: https://dasp.co/#item-4.

11. Manning A. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns: Arithmetic Over/Under Flows [Електронний ресурс] / Adrian

Manning. – 2018. – Режим доступу до ресурсу: https://blog.sigmaprime.io/solidity-security.html#ouflow.

12. Manning A. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns: Race Conditions / Front Running [Електронний ресурс] / Adrian Manning. – 2018. – Режим доступу до ресурсу: https://blog.sigmaprime.io/solidity-security.html#race-conditions.

13. Manning A. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns: Denial Of Service (DOS) [Електронний ресурс] / Adrian Manning. – 2018. – Режим доступу до ресурсу: https://blog.sigmaprime.io/solidity-security.html#dos.

14. Manning A. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns: Entropy Illusion [Електронний ресурс] / Adrian Manning. – 2018. – Режим доступу до ресурсу: https://blog.sigmaprime.io/solidity-security.html#SP-6.

15. Manning A. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns: Uninitialised Storage Pointers [Електронний ресурс] / Adrian Manning. – 2018. – Режим доступу до ресурсу: https://blog.sigmaprime.io/solidity-security.html#storage .

16. Structure of a Contract [Електронний ресурс] – Режим доступу до ресурсу: https://docs.soliditylang.org/en/latest/structure-of-a-contract.html#function-modifiers.

17. Order of Precedence of Operators [Електронний ресурс] – Режим доступу до ресурсу: https://docs.soliditylang.org/en/latest/cheatsheet.html.

18. Solidity v0.8.0 Breaking Changes [Електронний ресурс] – Режим доступу до ресурсу: https://docs.soliditylang.org/en/v0.8.10/080-breaking-changes.html.

19. Пілова Д. П. Методичні вказівки до виконання економічної частини дипломного проекту для студентів спеціальності 125 Кібербезпека [Електронний ресурс] / Д. П. Пілова.

20. Методичні рекомендації до виконання кваліфікаційних робіт бакалаврів спеціальності 125 Кібербезпека [Електронний ресурс] / О. В.Герасіна, Д. С. Тимофєєв, О. В. Кручінін, Ю. А. Мілінчук – Режим доступу до ресурсу: http://bit.nmu.org.ua/ua/student/diplom/Метод_КРБ_125_2020.pdf.

ДОДАТОК А. Відомість матеріалів кваліфікаційної роботи

№	Формат	Найменування	Кількість листів	Примітка
1	А4	Реферат	2	
2	А4	Список умовних скорочень	1	
3	А4	Зміст	2	
4	А4	1 Розділ	23	
5	А4	2 Розділ	19	
6	А4	3 Розділ	14	
7	А4	4 Розділ	8	
8	А4	Висновки	2	
9	А4	Список літератури	3	
10	А4	Додаток А. Відомість матеріалів кваліфікаційної роботи	1	
11	А4	Додаток Б. Перелік документів на оптичному носії	1	
12	А4	Додаток В. Відгук керівника економічного розділу	1	
13	А4	Додаток Г. Відгук керівника	1	
14	А4	Додаток Ґ. Лістинг коду смарт-контракту IERC20.sol	1	
15	А4	Додаток Д. Лістинг коду смарт-контракту ERC20.sol	5	
16	А4	Додаток Е. Початковий лістинг коду смарт-контракту PPSD.sol	2	
17	А4	Додаток Є. Лістинг коду смарт-контракту PPSD.sol з урахуванням запропонованих рішень щодо протидії виявленим вразливостям.	3	
18	А4	Додаток Ж. Лістинг бібліотеки SafeMath.sol	4	

ДОДАТОК Б. Перелік документів на оптичному носії

1 Пояснювальна записка.docx

2 Пояснювальна записка.pdf

3 Презентація.pptx

ДОДАТОК В. Відгук керівника економічного розділу

Відгук керівника економічного розділу:

____

____

____

____

____

____

____

____

____

____

____

____

____

Керівник розділу _______________ _________________

(підпис) (ініціали, прізвище)

ДОДАТОК Г. Відгук керівника

Керівник

ДОДАТОК Ґ. Лістинг коду смарт-контракту IERC20.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

interface IERC20 {
        function totalSupply() external view returns (uint256);
        function decimals() external view returns (uint8);
        function symbol() external view returns (string memory);
        function name() external view returns (string memory);
        function balanceOf(address account) external view returns (uint256);
        function transfer(address recipient, uint256 amount) external returns (bool);
        function allowance(address _owner, address spender) external view returns (
        uint256);
        function approve(address spender, uint256 amount) external returns (bool);
        function transferFrom(address sender, address recipient, uint256 amount)
        external returns (bool);
        event Transfer(address indexed from, address indexed to, uint256 value);
        event Approval(address indexed owner, address indexed spender,
        uint256 value);
}
```

ДОДАТОК Д. Лістинг коду смарт-контракту ERC20.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

import "./IERC20.sol";

contract ERC20 is IERC20 {
	mapping (address => uint256) private _balances;
	mapping (address => mapping (address => uint256)) private _allowances;

	uint256 private _totalSupply;
	string private _name;
	string private _symbol;

	constructor (string memory name_, string memory symbol_) {
		_name = name_;
		_symbol = symbol_;
	}

	function _msgSender() internal view returns (address) {
		return msg.sender;
	}

	function name() public view virtual override returns (string memory) {
		return _name;
	}
```

```
function symbol() public view virtual override returns (string memory) {
        return _symbol;
}

function decimals() public view virtual override returns (uint8) {
        return 18;
}

function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
}

function balanceOf(address account) public view virtual override returns (
uint256) {
        return _balances[account];
}

function transfer(address recipient, uint256 amount) public virtual override
returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
}

function allowance(address owner, address spender) public view virtual
override returns (uint256) {
        return _allowances[owner][spender];
}
```

```
function approve(address spender, uint256 amount) public virtual override
returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
}

function transferFrom(address sender, address recipient, uint256 amount)
public virtual override returns (bool) {
        _transfer(sender, recipient, amount);
        uint256 currentAllowance = _allowances[sender][_msgSender()];
        require(currentAllowance >= amount, "ERC20: transfer amount exceeds
        allowance");
        _approve(sender, _msgSender(), currentAllowance - amount);
        return true;
}

function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender]
        + addedValue);
        return true;
}

function decreaseAllowance(address spender, uint256 subtractedValue) public
virtual returns (bool) {
        uint256 currentAllowance = _allowances[_msgSender()][spender];
        require(currentAllowance >= subtractedValue, "ERC20: decreased
        allowance below zero");
```

```
        _approve(_msgSender(), spender, currentAllowance - subtractedValue);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal
    virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
        uint256 senderBalance = _balances[sender];
        require(senderBalance >= amount, "ERC20: transfer amount exceeds
        balance");
        _balances[sender] = senderBalance - amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");
        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);
    }

    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");
        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds
        balance");
        _balances[account] = accountBalance - amount;
```

```
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);
    }

    function _approve(address owner, address spender, uint256 amount) internal
    virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");
        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
}
```

ДОДАТОК Е. Початковий лістинг коду смарт-контракту PPSD.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

import "./ERC20.sol";

contract ppsd is ERC20 {
	uint private price = 370000000000000;
	address payable forwardingAddress =
	payable(0x3B06b969289E9c04BFe39D3A394c40d45A1B2018);
	address private owner;

	constructor() ERC20("PPSD", "PPSD") {
		owner = tx.origin;
		_mint(owner, 1_000_000 * (10 ** uint256(decimals())));
		_mint(address(this), 1_000_000 * (10 ** uint256(decimals())));
	}

	function changePrice(uint256 newPrice) public{
		price = newPrice;
	}

	function changeForwardingAddress(address newAddress) public {
		forwardingAddress = payable(newAddress);
	}

	function changeOwner(address newOwner) public {
```

```
            owner = newOwner;
        }

        function retrieveTokens(uint256 amount) public {
            require(tx.origin == owner);
            _transfer(address(this), tx.origin, amount);
        }

        receive() external payable {
            uint40 amount = msg.value * (10 ** decimals()) / price;
            _transfer(address(this), tx.origin, amount);
            forwardingAddress.send(address(this).balance);
        }
}
```

ДОДАТОК Є. Лістинг коду смарт-контракту PPSD.sol з урахуванням запропонованих рішень щодо протидії виявленим вразливостям

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

import "./ERC20.sol";

contract ppsd is ERC20 {
    uint private price = 370000000000000;
    address payable forwardingAddress =
    payable(0x3B06b969289E9c04BFe39D3A394c40d45A1B2018);
    address private owner;

    constructor() ERC20("PPSD", "PPSD") {
        owner = msg.sender;
        _mint(owner, 1_000_000 * (10 ** uint256(decimals())));
        _mint(address(this), 1_000_000 * (10 ** uint256(decimals())));
    }

    modifier onlyOwner {
        require(msg.sender == owner, "This operation is only allowed to the
        owner.");
        _;
    }

    function isValidAddress(address addr) internal view returns (bool) {
        uint256 size;
```

```
        assembly { size := extcodesize(addr) }
        return size == 40;
}

function changePrice(uint256 newPrice) external onlyOwner {
        require(newPrice != 0, "Token price cannot be 0.");
        price = newPrice;
}

function changeForwardingAddress(address newAddress) external
onlyOwner {
        require(newAddress != address(0), "New address can't be null.");
        require(isValidAddress(newAddress), "Invalid address specified");
        forwardingAddress = payable(newAddress);
}

function changeOwner(address newOwner) external onlyOwner {
        require(newOwner != address(0), "New address can't be null.");
        require(isValidAddress(newAddress), "Invalid address specified");
        owner = newOwner;
}

function retrieveTokens() external onlyOwner {
        uint256 amount = balanceOf(address(this));
        _transfer(address(this), owner, amount);
}

receive() external payable{
```

```
            uint256 amount = msg.value * (10 ** decimals()) / price;
            _transfer(address(this), msg.sender, amount);
            forwardingAddress.transfer(address(this).balance);
        }
}
```

ДОДАТОК Ж. Лістинг бібліотеки SafeMath.sol

```
pragma solidity ^0.6.0;

library SafeMath {
	/*
	* @dev Returns the addition of two unsigned integers, reverting on
	* overflow.
	*/
	function add(uint256 a, uint256 b) internal pure returns (uint256) {
		uint256 c = a + b;
		require(c >= a, "SafeMath: addition overflow");
		return c;
	}

	/*
	* @dev Returns the subtraction of two unsigned integers, reverting on
	* overflow (when the result is negative).
	*/
	function sub(uint256 a, uint256 b) internal pure returns (uint256) {
		return sub(a, b, "SafeMath: subtraction overflow");
	}

	/*
	* @dev Returns the subtraction of two unsigned integers, reverting with custom
	* message on overflow (when the result is negative).
	*/
	function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
		require(b <= a, errorMessage);
```

```
        uint256 c = a - b;
        return c;
}

/*
* @dev Returns the multiplication of two unsigned integers, reverting on
* overflow.
*/
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
                return 0;
        }
        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");
        return c;
}

/*
* @dev Returns the integer division of two unsigned integers. Reverts on
* division by zero. The result is rounded towards zero.
*/
function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
}

/*
* @dev Returns the integer division of two unsigned integers. Reverts with
* custom message on division by zero. The result is rounded towards zero.
```

```
        */
        function div(uint256 a, uint256 b, string memory errorMessage) internal pure
        returns (uint256) {
                require(b > 0, errorMessage);
                uint256 c = a / b;
                assert(a == b * c + a % b);
                return c;
        }

        /*
        * @dev Returns the remainder of dividing two unsigned integers. (unsigned
        * integer modulo),
        * Reverts when dividing by zero.
        */
        function mod(uint256 a, uint256 b) internal pure returns (uint256) {
                return mod(a, b, "SafeMath: modulo by zero");
        }

        /*
        * @dev Returns the remainder of dividing two unsigned integers. (unsigned
        * integer modulo),
        * Reverts with custom message when dividing by zero.
        */
        function mod(uint256 a, uint256 b, string memory errorMessage) internal pure
        returns (uint256) {
                require(b != 0, errorMessage);
                return a % b;
        }
}
```