

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Навчально-науковий інститут телекомунікаційних систем

Кафедра телекомунікацій

До захисту допущено:

В.о.завідувача кафедри

_____ Сергій КРАВЧУК

«__» _____ 2026 р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Інженерія та програмування
інфокомунікацій»**

спеціальності 172 «Телекомунікації та радіотехніка»

**на тему: «Розробка корпоративного сайту з використанням фреймворку
Django та оцінка рівня його вразливості щодо кібератак»**

Виконав:

Студент IV курсу, групи ТЗ-22

Чернець Ілля Олександрович _____

Керівник:

Доцент кафедри ТК НН ІТС, к.т.н., доцент,

Трубаров І.В. _____

Консультант з назва розділу:

Старший викладач кафедри ТК НН ІТС, к.ф.-м.н.,

Руренко О. Г. _____

Рецензент:

Доцент кафедри ІТТ НН ІТС, к.т.н., доцент,

Правило В.В. _____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студент _____

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут телекомунікаційних систем
Кафедра телекомунікацій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 172 «Телекомунікації та радіотехніка»

Освітньо-професійна програма «Інженерія та програмування інфокомунікацій»

ЗАТВЕРДЖУЮ

В.о.завідувача кафедри

_____ Сергій КРАВЧУК

«___» _____ 2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

Чернецю Іллі Олександровичу

1. Тема роботи «Розробка корпоративного сайту з використанням фреймворку Django та оцінка рівня його вразливості щодо кібератак», керівник роботи Трубаров Ігор Володимирович, к.т.н., доцент, затверджені наказом по університету від «26» травня 2026 р. № 1912-с.

2. Термін подання студентом роботи 10 червня 20265 р.

3. Вихідні дані до роботи: вимоги до розроблення корпоративного вебсайту телекомунікаційного підприємства; функціональні вимоги до підсистем керування завданнями, моніторингу мережевих вузлів і захищеного зберігання конфігураційних даних; мова програмування Python, вебфреймворк Django, СУБД PostgreSQL, технології HTML, CSS та JavaScript; рекомендації OWASP щодо захисту вебзастосунків; методи рольового й об'єктного контролю доступу, серверної валідації, криптографічного захисту даних і тестування вебзастосунку на вразливості.

4. Зміст роботи: 1. Провести аналіз призначення корпоративних вебсайтів у телекомунікаційній сфері та актуальних загроз за методологією OWASP. 2. Обґрунтувати вибір фреймворку Django, СУБД PostgreSQL і технологічного стеку для реалізації корпоративного вебсайту. 3. Спроектувати програмну архітектуру системи та структуру бази даних. 4. Розробити підсистеми управління завданнями, моніторингу мережевих вузлів і захищеного зберігання

інфраструктурних секретів. 5. Реалізувати механізми рольового й об'єктного контролю доступу, серверної валідації та криптографічного захисту даних. 6. Провести аналіз інформаційної безпеки розробленого вебзастосунку та перевірити його стійкість до типових вебатак. 7. Змодельовати експлуатацію виявлених вразливостей у контрольованому середовищі. 8. Розробити програмні заходи усунення вразливостей і провести повторне тестування системи.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо):

Слайд №1 Мета, об'єкт, предмет і завдання дослідження

Слайд №2 Аналіз загроз інформаційній безпеці корпоративних вебзастосунків

Слайд №3 Обґрунтування технологічного стеку та архітектури корпоративного вебсайту

Слайд №4 Структура програмної системи та UML-модель бази даних

Слайд №5 Реалізація підсистеми управління завданнями та контролю доступу

Слайд №6 Реалізація підсистем моніторингу мережевих вузлів і захищеного зберігання конфігурацій

Слайд №7 Методика та сценарії тестування інформаційної безпеки розробленої системи

Слайд №8 Моделювання ланцюга експлуатації виявлених вразливостей

Слайд №9 Усунення вразливостей і результати повторного тестування

Слайд №10 Практичне значення та основні результати роботи

6. Консультанти розділів роботи^{1*}

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ №1	Ст. викладач кафедри ТК, к.ф.-м.н. Руренко О. Г.	16.04.2026	18.04.2026
Розділ №2	Ст. викладач кафедри ТК, к.ф.-м.н. Руренко О. Г.	05.05.2026	07.05.2026
Розділ №3	Ст. викладач кафедри ТК, к.ф.-м.н. Руренко О. Г.	21.05.2026	22.05.2026

7. Дата видачі завдання 16.04.2026

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Обґрунтування актуальності теми та постановка завдань роботи.	16.04.2026 - 20.04.2026	Виконано
2	Пошук і аналіз літературних джерел за темою роботи.	20.04.2026 - 29.04.2026	Виконано

^{1*} Консультантом не може бути зазначено керівника дипломної роботи.

3	Дослідження загроз веб застосунків і підготовка розділу №1.	29.04.2026 - 03.05.2026	Виконано
4	Проектування архітектури та бази даних корпоративного сайту.	07.05.2026 - 10.05.2026	Виконано
5	Розроблення програмних підсистем і підготовка розділу №2.	10.05.2026 - 18.05.2026	Виконано
6	Тестування інформаційної безпеки та моделювання експлуатації вразливостей.	18.05.2026 - 21.05.2026	Виконано
7	Усунення вразливостей, повторне тестування та підготовка розділу №3.	21.05.2026 - 05.06.2026	Виконано
8	Оформлення пояснювальної записки дипломної роботи.	05.06.2026 - 08.06.2026	Виконано
9	Підготовка презентації, ілюстративного матеріалу та доповіді до захисту.	08.06.2026 - 13.06.2026	Виконано

Студент

Ілля ЧЕРНЕЦЬ

Керівник

Ігор ТРУБАРОВ

РЕФЕРАТ

Текстова частина дипломної роботи містить: 85 сторінок, 20 рисунків, 10 таблиць, 25 лістингів та 26 джерел.

У роботі розглянуто розроблення корпоративного веб сайту телекомунікаційного підприємства на базі Django. Реалізовано підсистеми управління завданнями, моніторингу мережевих вузлів і захищеного зберігання інфраструктурних секретів. Проведено перевірку контролю доступу, серверної валідації та захисту від основних веб атак.

Метою роботи є проєктування захищеного корпоративного вебсайту, оцінювання рівня його інформаційної безпеки та програмне усунення виявлених вразливостей. Змодельовано експлуатацію OS Command Injection і розкриття криптографічного ключа. Реалізовані захисні зміни пройшли повторне тестування.

Ключові слова: корпоративний вебсайт, Django, кібербезпека, PostgreSQL, OWASP, контроль доступу, OS Command Injection, Fernet.

ABSTRACT

The thesis comprises 85 pages, 20 figures, 10 tables, 25 code listings, and 26 references.

The thesis considers the development of a Django-based corporate website for a telecommunications company. Task management, network node monitoring, and secure infrastructure secret storage subsystems were implemented. Access control, server-side validation, and protection against common web attacks were tested.

The purpose of the thesis is to design a secure corporate website, assess its information security, and eliminate the identified vulnerabilities. The exploitation of OS command injection and cryptographic key disclosure was simulated. The implemented security improvements were successfully retested.

Keywords: corporate website, Django, cybersecurity, PostgreSQL, OWASP, access control, OS command injection, Fernet.

ЗМІСТ

ВСТУП.....	10
РОЗДІЛ 1 ТЕОРЕТИКО-АНАЛІТИЧНІ ОСНОВИ РОЗРОБКИ ТА КІБЕРЗАХИСТУ КОРПОРАТИВНИХ САЙТІВ	13
1.1 Роль та специфіка корпоративних сайтів у сфері телекомунікацій.....	13
1.2 Класифікація та аналіз вразливостей веб-додатків за методологією OWASP	14
1.3 Огляд та порівняльний аналіз сучасних веб-фреймворків у контексті кібербезпеки.....	18
1.4 Обґрунтування вибору технологічного стеку	20
1.5 Системна архітектура та аналіз базових компонентів фреймворку Django	21
1.6 Життєвий цикл HTTP-запиту на рівні внутрішніх механізмів Django..	24
Висновки	25
РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ КОРПОРАТИВНОГО САЙТУ	27
2.1 Проєктування логічної структури бази даних (UML-модель)	28
2.2 Ініціалізація проєкту та конфігурація середовища розробки	29
2.3 Програмна реалізація модуля управління завданнями	32
2.4 Програмна реалізація підсистеми моніторингу мережевих вузлів	44
2.5 Програмна реалізація підсистеми безпечного зберігання інфраструктурних конфігурацій (Vault)	49
Висновки	57
РОЗДІЛ 3 АНАЛІЗ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ТА ПРАКТИЧНА ЕКСПЛУАТАЦІЯ ВРАЗЛИВОСТЕЙ	59
3.1 Методика та умови проведення практичного дослідження.....	59
3.2 Перевірка реалізованих механізмів захисту	62
3.3 Моделювання ланцюга експлуатації вразливостей	73
3.4 Усунення виявлених вразливостей і повторна перевірка	77
Висновки	81
ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ	82

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	84
----------------------------------	----

ПЕРЕЛІК СКОРОЧЕНЬ

AES	Advanced Encryption Standard
ASGI	Asynchronous Server Gateway Interface
CSRF	Cross-Site Request Forgery
IDOR	Insecure Direct Object Referenc
MVT	Model-View-Template
NOC	Network Operations Center
ORM	Object-Relational Mapping
OWASP	Open Worldwide Application Security Project
RBAC	Role-Based Access Control
WSGI	Web Server Gateway Interface

ВСТУП

Актуальність теми. На сучасному етапі розвитку інфокомунікацій спостерігається стрімка цифровізація бізнес-процесів, що супроводжується активним впровадженням веборієнтованих систем корпоративного управління. Підприємства телекомунікаційної та ІТ-галузей все частіше використовують внутрішні ресурси для координації роботи персоналу, моніторингу мережевої інфраструктури та обміну конфіденційною технічною інформацією. Проте швидкість розробки таких систем часто превалює над вимогами безпеки, що призводить до появи критичних вразливостей. Згідно зі звітами міжнародного консорціуму OWASP, внутрішні корпоративні додатки часто стають найслабшою ланкою в системі кіберзахисту. Використання сучасних високорівневих фреймворків, зокрема Django, забезпечує базовий рівень захисту від типових атак, але не гарантує стійкості проти логічних вразливостей, помилок у рольових моделях доступу чи специфічних атак на мережеві інтерфейси. Для телеком-операторів, де витік облікових даних, конфігурацій маршрутизаторів чи криптографічних ключів може призвести до компрометації всієї магістральної мережі, розробка захищених систем зберігання секретів та засобів моніторингу є критично важливим завданням. Таким чином, тема роботи, спрямована на поєднання процесу розробки корпоративного сайту з глибокою оцінкою та усуненням його вразливостей, є актуальною та практично значущою.

Мета і завдання дослідження. Метою роботи є проєктування та програмна реалізація захищеного корпоративного сайту на базі фреймворку Django, а також проведення комплексного аналізу його інформаційної безпеки з подальшим програмним усуненням виявлених архітектурних вразливостей.

Для досягнення поставленої мети в роботі поставлено такі завдання:

1. Провести аналіз сучасних загроз для вебдодатків згідно з методологією OWASP Top 10:2025 та визначити специфіку їх прояву в інфокомунікаційних системах.

2. Спроекувати архітектуру бази даних та реалізувати програмні модулі управління завданнями, мережевого моніторингу й захищеного зберігання інфраструктурних секретів.
3. Розробити криптографічну підсистему «Vault» для захищеного зберігання інфраструктурних секретів.
4. Здійснити практичну оцінку рівня захищеності розробленого додатка шляхом моделювання кібератак типу OS Command Injection та експлуатації криптографічних збоїв.
5. Розробити та імплементувати механізми захисту на рівні вихідного коду (санітизація вводу, безпечний виклик підпроцесів, ізоляція ключів) та підтвердити їх ефективність.

Об'єкт дослідження – процес розробки та захисту корпоративного сайту для підтримки внутрішніх процесів телекомунікаційного підприємства.

Предмет дослідження – програмні механізми контролю доступу, валідації вхідних даних, безпечного виклику системних утиліт та управління криптографічними ключами у веб додатку на базі Django.

Методи дослідження. У роботі використано методи об'єктно-орієнтованого програмування для побудови логіки додатка; методи реляційного моделювання для проєктування бази даних PostgreSQL; методи white-box аналізу вихідного коду та динамічного моделювання атак для перевірки стійкості реалізованих механізмів захисту; методи симетричного шифрування (стандарт AES-128 у режимі CBC) для захисту даних у стані спокою.

Наукова новизна отриманих результатів полягає в удосконаленні підходу до побудови корпоративного сайту шляхом поєднання модулів мережевої діагностики, рольового контролю доступу та криптографічного сховища секретів із подальшою перевіркою їх стійкості до типових вебвразливостей.

Практичне значення отриманих результатів полягає у створенні прототипу корпоративного сайту та демонстрації повного циклу безпечної

розробки: від реалізації функціональних модулів до виявлення, експлуатації й усунення вразливостей на рівні вихідного коду.

Можливі галузі застосування результатів роботи. Розроблений програмний комплекс орієнтований на практичне впровадження у внутрішню мережеву інфраструктуру інтернет-провайдерів, дата-центрів та ІТ-компаній для автоматизації роботи інженерів Центрів управління мережею та безпечного адміністрування телекомунікаційного обладнання.

РОЗДІЛ 1

ТЕОРЕТИКО-АНАЛІТИЧНІ ОСНОВИ РОЗРОБКИ ТА КІБЕРЗАХИСТУ КОРПОРАТИВНИХ САЙТІВ

1.1 Роль та специфіка корпоративних сайтів у сфері телекомунікацій

У сучасних умовах функціонування інфокомунікаційних мереж корпоративні сайти перестали бути просто інформаційними ресурсами і перетворилися на складні інтегровані середовища управління бізнес-процесами. Для підприємств галузі телекомунікацій та радіотехніки такі системи виконують роль центрального вузла, що об'єднує технічний персонал, адміністраторів мереж та менеджмент.

Специфіка телекомунікаційної сфери вимагає від корпоративних систем підтримки специфічного функціоналу:

1. Моніторинг мережевої інфраструктури: можливість швидкої діагностики доступності серверів, маршрутизаторів та вузлів зв'язку безпосередньо через веб-інтерфейс.
2. Управління конфігураціями та секретами: безпечне зберігання та передача API-ключів, SSL-сертифікатів та паролів доступу до мережевого обладнання.
3. Координація технічних робіт: системи управління завданнями, що дозволяють розподіляти заявки на ремонт або обслуговування ліній зв'язку між інженерами.

За результатами аналізу функціональних вимог визначено, що розроблювана система повинна поєднувати засоби координації технічних робіт, мережевої діагностики та контрольованого зберігання інфраструктурних секретів. Таке поєднання функцій виходить за межі типового інформаційного порталу, оскільки передбачає взаємодію вебдодатка із системними мережевими утилітами та оброблення даних із різними рівнями конфіденційності. Тому для реалізації поставлених завдань обрано спеціалізований корпоративний сайт,

архітектуру якого можна адаптувати до рольової моделі підприємства та визначених сценаріїв кібербезпеки.

1.2 Класифікація та аналіз вразливостей веб-додатків за методологією OWASP

Кібербезпека корпоративного сайту є критичним фактором, оскільки успішна атака на таку систему може призвести до компрометації не лише даних персоналу, а й усієї мережевої інфраструктури компанії. Для систематизації загроз у роботі використано міжнародний стандарт OWASP Top 10, який визначає найбільш критичні ризики для веб-додатків. Узагальнену модель формування ризику наведено на рис. 1.1

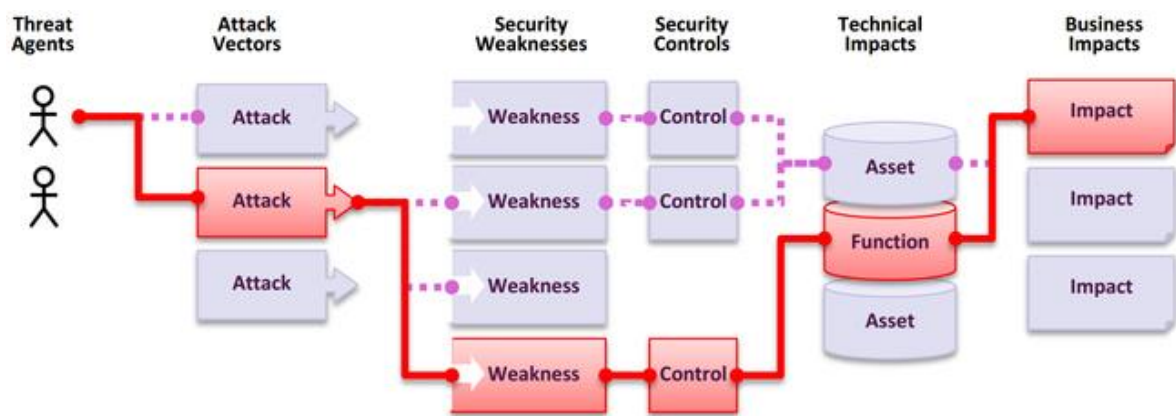


Рис. 1.1 – Шлях агенту загрози до завдання шкоди бізнесу

Джерело: адаптовано автором за матеріалами OWASP [1].

Поверхня атаки веб додатку охоплює сукупність доступних зловмиснику точок взаємодії з його компонентами. Рівень ризику для кожної такої точки визначається не лише технічною можливістю експлуатації вразливості, а й імовірністю відсутності належного захисного контролю та потенційним впливом інциденту на діяльність організації [1]. Для розроблюваного корпоративного сайту до критичних точок взаємодії належать механізми авторизації, форми введення мережевих адрес і підсистема зберігання інфраструктурних секретів.

У контексті розробки на Python/Django та специфіки телеком-інструментів, найбільш небезпечними є наступні категорії вразливостей:

1. Недоліки контролю доступу

У редакції OWASP Top 10:2025 порушення контролю доступу зберігає першу позицію; відповідні недоліки були виявлені в усіх застосунках, охоплених набором тестових даних OWASP [2]. Для цієї роботи принциповим є не саме місце категорії у рейтингу, а характер її прояву: сервер приймає запит від автентифікованого користувача, проте не перевіряє, чи має він право виконувати дію над конкретним об'єктом. У такому разі наявність активного сеансу не гарантує правомірності операції.

У розроблюваній системі цей ризик стосується насамперед об'єктів Task і Secret. Підміна ідентифікатора в URL-адресі може дати змогу звернутися до чужого завдання або секрету, якщо представлення перевіряє лише факт входу до системи. Тому авторизацію необхідно реалізовувати на сервері для кожної операції читання, зміни й видалення, поєднуючи принцип заборони за замовчуванням із перевіркою ролі користувача та належності об'єкта [2; 3].

2. Криптографічні збої

Категорія A04:2025 охоплює ситуації, у яких конфіденційні дані залишаються незахищеними через відсутність шифрування, застосування неналежних алгоритмів або помилки керування криптографічними ключами [4]. Для корпоративного сайту важливо розрізняти захист даних під час передавання та захист даних, збережених у базі. Використання HTTPS обмежує перехоплення мережевого трафіку, але саме по собі не захищає вміст резервної копії або дампа бази даних.

У підсистемі Vault основним ризиком є не лише вибір алгоритму, а й спосіб зберігання ключового матеріалу. Якщо ключ розміщено у вихідному коді разом із логікою дешифрування, компрометація репозиторію або серверних файлів надає атакувальнику всі компоненти, необхідні для відновлення відкритих даних. Така реалізація відповідає класу CWE-321 і

суперечить принципу відокремлення криптографічного ключа від даних та програмного коду, що їх обробляє [4; 5]. Отже, у проєкті необхідно окремо оцінювати стійкість алгоритму Fernet і захищеність життєвого циклу ключа.

3. Ін'єкції

Ін'єкційна вразливість виникає тоді, коли зовнішні дані потрапляють до інтерпретатора в такій формі, що можуть змінити структуру запиту або системної команди. Її першопричиною є змішування даних із виконуваним синтаксисом, а не наявність окремого «небезпечного символу». Тому блокування обмеженого набору метасимволів не є достатнім захисним заходом [3; 6; 7].

Для розроблюваного сайту необхідно розмежувати два сценарії. Запити до PostgreSQL формуються через Django ORM, що за штатного використання параметризує значення і знижує ризик SQL-ін'єкції. Водночас модуль Network передає введену користувачем адресу системній утиліті, тому небезпечно формування одного командного рядка з `shell=True` створює окремий вектор OS Command Injection. Належний захист передбачає валідацію цільової адреси, передавання аргументів процесу списком і відмову від командної оболонки, якщо її функції не потрібні [3; 6].

4. Міжсайтовий скриптинг

Міжсайтовий скриптинг є одним із найнебезпечніших векторів атак на клієнтську частину веб-додатків, спрямованим безпосередньо на користувачів системи. Ця вразливість виникає в ситуаціях, коли сервер приймає неперевірені дані від користувача і відправляє їх назад у браузер без належної процедури валідації або контекстно-залежного екранування. Внаслідок цього браузер жертви інтерпретує впроваджене корисне навантаження, найчастіше це шкідливий JavaScript-код, як легітимну частину сторінки та виконує його в контексті довіреного сеансу.

У корпоративних мережеских інтерфейсах та системах управління інфраструктурою наслідки такої атаки можуть мати критичний характер. Найвищу загрозу становить Stored XSS. Якщо зловмиснику вдається

впровадити такий скрипт у спільні модулі системи – наприклад, у поле опису завдання (Task Manager) або в журнал моніторингу, – цей код буде автоматично виконуватися в браузері кожного інженера чи адміністратора, який відкриє відповідну сторінку. Це створює пряму загрозу прихованого викрадення сесійних ідентифікаторів, токенів безпеки або навіть виконання несанкціонованих адміністративних дій, таких як додавання нових користувачів від імені привілейованого облікового запису.

Захист від XSS доцільно будувати як сукупність незалежних механізмів. Шаблонізатор Django автоматично екранує більшість змінних у HTML-контексті, однак цей механізм не усуває ризику в разі використання позначення безпечного рядка, вимкнення екранування або передавання даних до іншого контексту, наприклад JavaScript [8; 9]. Додатковим рівнем є політика Content Security Policy, яка обмежує джерела завантаження та виконання сценаріїв у браузері [10]. Прапорець HttpOnly для сесійного cookie зменшує можливість зчитування його значення клієнтським сценарієм, але не замінює коректного кодування вихідних даних і налаштування CSP [8].

З метою систематизації теоретичних відомостей та формування базису для проведення подальшого практичного дослідження безпеки, у таблиці 1.1 наведено відповідності між класифікатором глобальних загроз OWASP Top 10, їх потенційним проявом у розроблюваному корпоративному сайті та методами їх практичної перевірки.

Таблиця 1.1

Відповідність загроз OWASP архітектурі розроблюваного корпоративного сайту

Категорія за OWASP Top 10	Потенційний прояв у корпоративному сайті	Цільовий модуль системи	Спосіб перевірки та експлуатації
A01:2025 - Broken Access Control	Несанкціоноване редагування чужих завдань або доступ до прихованих секретів через маніпуляцію ID	Vault, Task Manager	Спроба прямого доступу до об'єкта (IDOR) в обхід рольової моделі (RBAC)
A04:2025 - Cryptographic Failures	Зберігання криптографічних ключів у вихідному коді, відсутність ізоляції середовища	Vault (модель Secret)	Ексфільтрація вихідного коду та офлайн-дешифрування БД
A05:2025 - Injection	Впровадження системних команд через несанітизовані поля IP-адрес	Network	Ін'єкція метасимволів ОС (; ls -la) у параметри утиліти ping
A02:2025 - Security Misconfiguration	Залишення режиму зневадження (Debug) або недостатня фільтрація виводу помилок	Ядро системи (Django)	Аналіз відповідей сервера під час введення невалідних структур даних

1.3 Огляд та порівняльний аналіз сучасних веб-фреймворків у контексті кібербезпеки

Вибір базового технологічного стеку для розробки корпоративного сайту вимагає збалансованого підходу до питань безпеки, швидкості розробки та простоти масштабування. Для обґрунтування архітектурного рішення було проведено порівняльний аналіз трьох популярних екосистем: Express.js

(Node.js), Spring Boot (Java) та Django (Python) за п'ятьма ключовими критеріями.

- 1. Автентифікація та авторизація.** У Spring-екосистемі відповідні механізми реалізуються через Spring Security, який надає розвинені засоби конфігурації правил доступу, але потребує окремого налаштування ланцюга фільтрів і політик [11]. Express.js не визначає єдиної штатної моделі автентифікації, тому її склад залежить від обраних сторонніх пакетів і рішень розробника [12]. Django постачається з моделлю користувача, сесіями, дозволами та адміністративним інтерфейсом, що відповідає вимогам цього проєкту [8].
- 2. Робота з базою даних.** Django ORM і JPA/Hibernate дають змогу формувати типові запити через об'єктні API. Для Django офіційна документація прямо вказує на параметризацію запитів QuerySet; винятком залишаються випадки використання необробленого SQL або небезпечного конструювання запиту [8]. В Express.js спосіб доступу до бази даних визначається додатково обраним драйвером або ORM.
- 3. Вебзахист за замовчуванням.** Django містить middleware для CSRF-захисту та механізм автоматичного екранування шаблонних змінних [8]. У Spring аналогічні політики реалізуються засобами Spring Security [11]. Для Express.js значна частина захисної конфігурації формується на рівні застосунку та middleware; зокрема, Helmet використовується для встановлення HTTP-заголовків безпеки [13].
- 4. Криптографічна реалізація.** Порівнювати фреймворки за «рівнем криптографії» некоректно, оскільки практична стійкість залежить від конкретної бібліотеки, режиму роботи алгоритму та керування ключами. У цьому проєкті Python-бібліотеку cryptography обрано через наявність готової реалізації Fernet, яка поєднує шифрування і перевірку цілісності повідомлення [14].
- 5. Відповідність завданню.** Вибір Django зроблено не на підставі універсальної переваги у швидкості розробки, а через відповідність

конкретним вимогам: наявність ORM, вбудованої моделі користувачів, класових представлень, middleware та адміністративного інтерфейсу. Це скорочує обсяг допоміжного коду й дає змогу зосередити дослідження на об'єктному контролі доступу, безпечному запуску мережових утиліт і керуванні криптографічним ключем.

Для наочності результати проведеного аналізу зведено у таблицю 1.2.

Таблиця 1.2

Порівняльна характеристика веб фреймворків за критеріями безпеки та розробки

Критерій оцінки	Express.js (Node.js)	Spring Boot (Java)	Django (Python)
Архітектурний підхід	Мікро-фреймворк	Корпоративний моноліт	"Batteries-included"
Вбудований захист (CSRF, XSS)	Через сторонні пакети	Вбудовано, через Spring Security	Вбудовано, на рівні ядра
Система ORM	Зовнішні бібліотеки	JPA / Hibernate	Вбудована Django ORM

1.4 Обґрунтування вибору технологічного стеку

На основі проведеного аналізу архітектурних вимог та загроз кібербезпеці, для реалізації корпоративного сайту було обрано наступний технологічний стек:

- Мова програмування: Python 3. Вибір Python обґрунтований його лідерством у сфері інформаційної безпеки та мережевого програмування. Багата екосистема бібліотек, зокрема cryptography для реалізації AES-шифрування у Secret Manager та вбудовані модулі subprocess і socket для розробки Network Monitor робить Python 3 доцільним вибором для реалізації поставлених завдань.
- Веб-фреймворк: Django. Архітектура MVT дозволяє чітко розділити бізнес-логіку (Views), структуру бази даних (Models) та інтерфейс користувача (Templates). Наявність вбудованої системи автентифікації та

адміністрування значно прискорює розробку базового функціоналу, дозволяючи зосередити основну увагу на реалізації рольової моделі доступу RBAC та усуненні специфічних логічних вразливостей IDOR.

- СУБД: PostgreSQL. Оскільки сайт оперуватиме конфіденційними даними (зашифровані секрети, паролі співробітників, конфігурації мережі), необхідна надійна реляційна база даних, що повністю відповідає принципам ACID. PostgreSQL є стандартом де-факто для розгортання Django-додатків у виробничому середовищі, забезпечуючи високу продуктивність при обробці паралельних транзакцій та стійкість до збоїв.

Отже, вибір технологічного стеку здійснено з урахуванням трьох груп вимог: підтримки рольової моделі доступу, безпечної роботи з реляційними даними та можливості контрольованої інтеграції системних мережевих утиліт.

1.5 Системна архітектура та аналіз базових компонентів фреймворку Django

Архітектура Django розділяє опис структури даних, оброблення HTTP-запитів і формування інтерфейсу між моделями, представленнями та шаблонами. У навчальній і профільній літературі такий підхід часто позначають як Model–View–Template, однак для технічного аналізу важливішим є фактичний розподіл відповідальності між компонентами [15; 16]. Модель визначає структуру й правила роботи з даними, представлення реалізує прикладну логіку, а шаблон відповідає за формування HTML-відповіді.

HTTP-запит обробляється не одним компонентом, а послідовністю рівнів: серверним шлюзом WSGI або ASGI, middleware, URL-диспетчером, представленням і шаблонізатором. Така послідовність дає змогу визначити, на якому рівні має виконуватися кожна перевірка безпеки [8; 15].

Для детального розуміння процесів ініціалізації та функціонування розробленого корпоративного сайту необхідно провести аналіз базових конфігураційних компонентів, які автоматично генеруються під час розгортання структури проекту:

- *manage.py* – є командним інтерфейсом конкретного Django-проєкту та встановлює для команд керування значення *DJANGO_SETTINGS_MODULE*. [15].

Якщо проаналізувати вихідний код ядра Django, то робота *manage.py* базується на виклику методу *execute_from_command_line* з пакета *django.core.management*. Механізм у внутрішній архітектурі працює за такою схемою:

- Скрипт передає шлях до конфігураційного файлу у змінні оточення: *os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')*.
- Функція *execute_from_command_line(sys.argv)* парсить аргументи командного рядка.
- Ініціюється клас *ManagementUtility*, який динамічно завантажує підкоманди з вбудованих додатків (*sqlmigrate*, *migrate*) та користувацьких модулів.
- *wsgi.py* (Web Server Gateway Interface) – виступає точкою входу для синхронної взаємодії Django-дodatка з веб сервером за стандартом WSGI. Цей компонент є первинним шаром, який приймає сирі HTTP-запити від зовнішнього веб-сервера (Nginx/Apache), трансформує їх у внутрішній формат об'єктів Django (*WSGIRequest*) та передає далі по конвеєру обробки. [16]

При отриманні HTTP-запиту, WSGI-сервер викликає об'єкт *application*, який є екземпляром класу *WSGIHandler*, визначений у системному модулі *django.core.handlers.wsgi*. Цей об'єкт виконує наступні підпроцеси:

- Завантажує змінні середовища та конфігурацію проєкту.
- Конвертує сирий HTTP-запит від сервера у високорівневий об'єкт *WSGIRequest*.
- Передає об'єкт *request* через ланцюжок проміжного програмного забезпечення до системи маршрутизації.
- *asgi.py* (Asynchronous Server Gateway Interface) – сучасний стандарт інтерфейсу, інтегрований для забезпечення асинхронної архітектури

додатка [16]. На відміну від WSGI, цей модуль дозволяє системі оперувати довготривалими з'єднаннями, що є критично важливим для реалізації протоколів WebSocket, протоколів передачі потокових даних або черг завдань у телекомунікаційних мережах, де синхронна модель призводила б до блокування потоків.

У контексті корпоративного сайту, наявність `asgi.py` закладає архітектурний фундамент для інтеграції інструментів реального часу, наприклад, потокового оновлення дашбордів моніторингу мережі без перезавантаження сторінки. Реалізовано це таким чином, що об'єкт `application` ініціює `ASGIHandler` з модуля `django.core.handlers.asgi`, який керує асинхронними подіями через подієвий цикл Python.

- `settings.py` – центральний ядровий модуль конфігурації інформаційної системи [15]. У ньому визначаються глобальні параметри функціонування сайту: підключення до СКБД, активація модулів безпеки, наприклад, захист від CSRF, XSS, сесійні налаштування, реєстрація встановлених додатків (включаючи `network` та `vault`), а також керування статичними файлами та медіа-контентом.
- `urls.py` – диспетчер URL-адрес та маршрутизатор запитів. Використання маршрутизатора URL забезпечує централізоване зіставлення запитів із представленнями, що спрощує впровадження перевірок автентифікації, авторизації та контролю доступу на рівні серверної логіки. [15]

Django використовує клас `URLResolver` з модуля `django.urls.resolvers`. Робота маршрутизатора відбувається за алгоритмом:

1. Отримання URL-рядка з об'єкта запиту, наприклад, `/admin/`.
2. Послідовне порівняння рядка з регулярними виразами або паттернами з масиву `urlpatterns`.
3. У разі збігу, `URLResolver` повертає відповідну функцію або клас представлення (`View`) та розпарсені параметри, наприклад, ID об'єкта для перевірки на IDOR.

1.6 Життєвий цикл HTTP-запиту на рівні внутрішніх механізмів

Django

Аналіз захищеності корпоративного інфокомунікаційного сайту вимагає глибокого розуміння внутрішніх механізмів обробки HTTP-запитів у середовищі Django. Згідно з парадигмою *Defense in Depth*, безпека веб додатків не обмежується єдиним бар'єром, а реалізується на кожному етапі трансформації даних [17]. Життєвий цикл запиту в архітектурі фреймворку декомпонується на п'ять ключових фаз, кожна з яких виконує специфічну роль у загальній системі кіберзахисту:

1. **Фаза ініціалізації (WSGI/ASGI).** Оброблення запиту в Django починається на рівні WSGI або ASGI, де сервер передає фреймворку дані HTTP-запиту у стандартизованому форматі. На цьому рівні формується об'єкт *HttpRequest*, однак перевірка коректності HTTP-протоколу та протидія атакам на неоднозначність запитів залежать також від конфігурації фронтального вебсервера й сервера застосунку. Тому WSGI/ASGI слід розглядати як інтерфейс взаємодії компонентів, а не як окремий механізм кіберзахисту [15; 18].
2. **Фаза фільтрації (Middleware Security Control).** Далі запит проходить через налаштований ланцюг *middleware*. Окремі компоненти цього ланцюга виконують спеціалізовані функції: *SessionMiddleware* пов'язує запит із сесансом, *AuthenticationMiddleware* додає відомості про користувача, а *CsrfViewMiddleware* перевіряє CSRF-захист для небезпечних методів запиту. Ефективність цього рівня визначається складом і порядком *middleware* у *settings.py*; сама наявність механізму у фреймворку не гарантує правильності його конфігурації [8; 17].
3. **Фаза маршрутизації (URL Resolution).** URL-диспетчер послідовно зіставляє шлях запиту з елементами *urlpatterns* і передає знайденому представленню параметри маршруту [15]. Конвертери на зразок *<int:pk>*

обмежують формат параметра, але не перевіряють повноваження користувача щодо відповідного об'єкта.

4. **Фаза бізнес-логіки (View-Level Authorization & ORM).** У представленні виконується прикладна логіка та звернення до бази даних. Django ORM знижує ризик SQL-ін'єкцій завдяки параметризації штатних запитів, однак не захищає від помилок об'єктної авторизації та небезпечних системних викликів [8]. Саме тому в роботі окремо перевіряються функція *can_manage_task()*, правила доступу до Secret і передавання користувацьких даних до subprocess.
5. **Фаза рендерингу та формування відповіді.** На завершальному етапі результат передається шаблонізатору й формується *HttpResponse*. Автоматичне екранування шаблонних змінних знижує ризик XSS у стандартному HTML-контексті, проте розробник повинен уникати необґрунтованого використання *safe*, ручного формування HTML і вставлення неекранованих значень у JavaScript [8; 9]. Після формування відповіді middleware може додати заголовки безпеки, зокрема захист від вбудовування сторінки у фрейм.

Підсумовуючи аналіз життєвого циклу HTTP-запиту, можна констатувати, що внутрішні механізми Django забезпечують стійкий ешелонований захист базового рівня. Водночас фаза обробки бізнес-логіки залишається відкритою для складних архітектурних вразливостей, що формує чітке методичне підґрунтя для проведення практичної експлуатації та подальшої рефакторизації коду, представлених у наступних розділах кваліфікаційної роботи.

ВИСНОВКИ

1. Встановлено, що для корпоративних веб сайтів телекомунікаційних підприємств найбільш критичними є ризики порушення контролю доступу, ін'єкцій команд ОС та компрометації криптографічних ключів, оскільки ці ризики безпосередньо пов'язані з обробкою інфраструктурних даних і мережевою діагностикою.

2. Обґрунтовано доцільність використання Django як базового фреймворку завдяки наявності ORM, middleware-рівня, шаблонного autoescaping та вбудованих механізмів CSRF-захисту, однак показано, що ці механізми не усувають логічні й архітектурні вразливості прикладного коду.
3. Визначено, що подальша реалізація системи має бути зосереджена на трьох функціональних модулях: управлінні завданнями, мережевому моніторингу та захищеному сховищі секретів, оскільки саме вони формують основну поверхню атаки для практичного дослідження в Розділі 3.
4. За результатами порівняльного аналізу сучасних фреймворків обґрунтовано вибір стеку технологій: мови Python, фреймворку Django та СУБД PostgreSQL.

РОЗДІЛ 2

ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ КОРПОРАТИВНОГО САЙТУ

У даному розділі дипломної роботи розглядається процес архітектурного проєктування та безпосередньої програмної реалізації корпоративного сайту для підприємства телекомунікаційної галузі. Основною метою розробки є створення інтегрованого цифрового середовища, яке оптимізує внутрішню взаємодію інженерного та управлінського персоналу, забезпечує моніторинг мережевої інфраструктури та передбачає реалізацію механізмів захищеного зберігання конфіденційних інфраструктурних даних із використанням симетричного шифрування та рольового контролю доступу.

В межах розділу вирішуються такі системно-інженерні завдання:

- обґрунтування вибору технологічного стеку та аналіз структури бази даних під управлінням СКБД PostgreSQL;
- опис декомпозиції системи на ізольовані функціональні модулі в межах архітектурного шаблону MVT;
- розробка та налаштування підсистеми розмежування прав доступу на основі рольової моделі (RBAC);
- реалізація інтерактивного інструментарію низькорівневої діагностики мережевих вузлів (модуль Network);
- програмне моделювання підсистеми криптографічного захисту інфраструктурних конфігурацій та секретів у стані спокою (модуль Vault) з гранулярним контролем доступу на рівні об'єктів.

Особлива увага приділяється практичній імплементації архітектурних рішень, які, з одного боку, забезпечують високу швидкість обробки інформаційних потоків і гнучкість управління, а з іншого – містять заздалегідь визначені особливості реалізації, що виступатимуть об'єктом дослідження під час проведення практичного дослідження безпеки у наступній частині роботи.

2.1 Проктування логічної структури бази даних (UML-модель)

Основою будь-якого надійного веб-додатка є грамотно спроектована реляційна база даних. Для реалізації корпоративного сайту було розроблено архітектуру, що базується на чотирьох основних сутностях, які відображають ключові бізнес-процеси телекомунікаційної компанії: управління персоналом, контроль виконання завдань, безпечне зберігання конфігурацій та моніторинг мережевого обладнання.

На рис. 2.1 наведено UML-діаграму класів, яка відображає основні сутності системи, їх атрибути та зв'язки між модулями автентифікації, управління завданнями, мережевого моніторингу й криптографічного сховища.

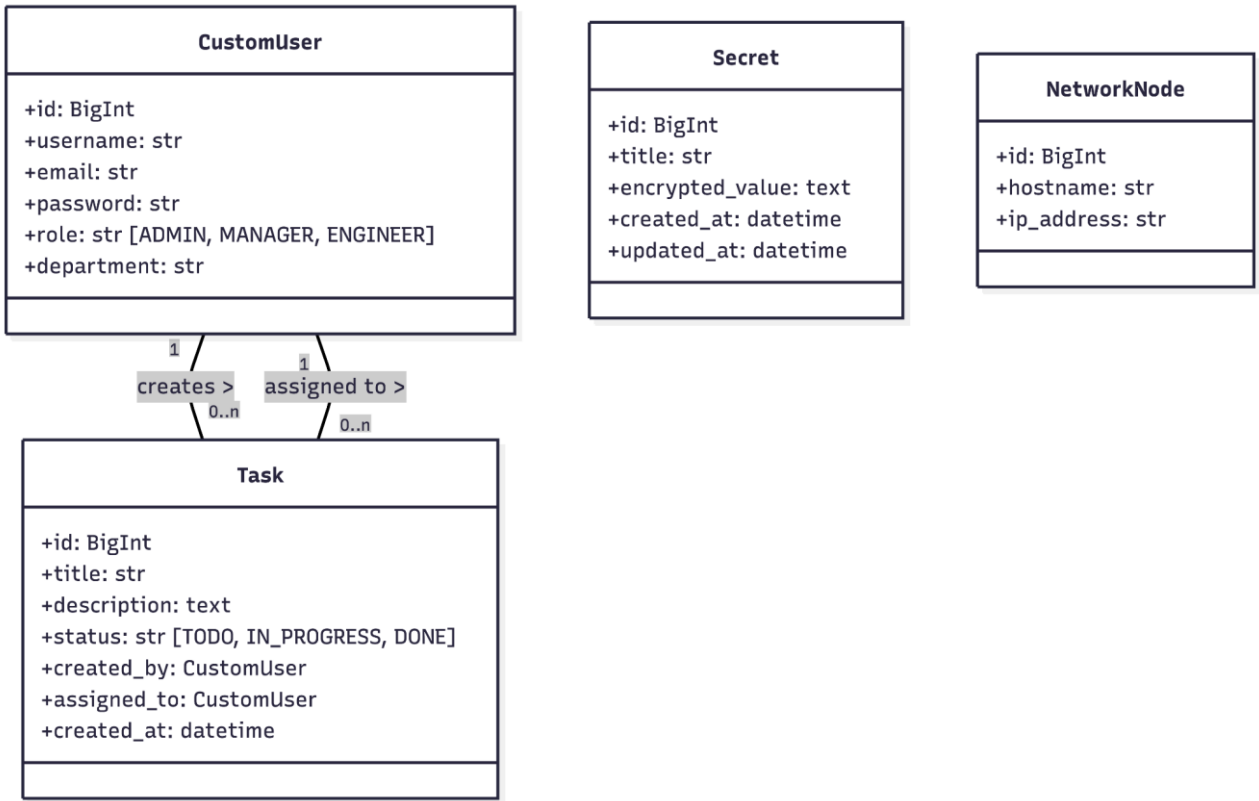


Рис. 2.1 – UML-діаграма класів проєкту

Для деталізації логічної структури бази даних та визначення ролі кожної сутності у забезпеченні інформаційної безпеки системи, у таблиці 2.1 наведено розширений архітектурний опис ключових реляційних моделей.

Таблиця 2.1

Опис ключових сутностей бази даних та їх роль у безпеці системи

Назва моделі	Призначення в системі	Ключові поля	Зв'язки	Роль в архітектурі безпеки
CustomUser	Управління профілями, автентифікація та авторизація персоналу.	username, password, role, department	1:N до Task, 1:N до Secret	Центральний вузол перевірки повноважень (RBAC). Визначає глобальний рівень доступу до функціональних підсистем.
Task	Ведення обліку інфраструктурних завдань та моніторинг їх виконання.	title, status, priority, deadline	N:1 до CustomUser (через created_by та assigned_to)	Об'єкт імплементації гранулярного контролю (Object-Level Permissions) для запобігання атакам класу IDOR.
NetworkNode	Інвентаризація корпоративного телекомунікаційного обладнання.	hostname, ip_address	Ізольована сутність	Знижує ризики ін'єкцій системних команд шляхом використання попередньо валідованих IP-адрес із БД.
Secret	Захищене зберігання криптографічних конфігурацій та паролів.	name, encrypted_value, access_type	N:1 до CustomUser	Реалізує концепцію захисту даних у стані спокою за допомогою алгоритму симетричного шифрування.

2.2 Ініціалізація проєкту та конфігурація середовища розробки

Процес програмної реалізації корпоративного сайту розпочався з налаштування ізольованого середовища розробки. Це є стандартом, що дозволяє уникнути конфліктів версій бібліотек під час розгортання проєкту на робочих серверах.

Етап 1. Створення віртуального середовища.

Для ізоляції залежностей було ініціалізовано віртуальне середовище Python за допомогою вбудованого модуля `venv`:

```
python -m venv .venv
source .venv/bin/activate
```

Після активації середовища було встановлено ядро веб-додатка – фреймворк Django версії 5.2 LTS, що гарантує отримання оновлень безпеки протягом тривалого часу.

Етап 2. Ініціалізація Django-проєкту та структура директорій.

За допомогою утиліти командного рядка `django-admin` було згенеровано базовий каркас проєкту:

```
django-admin startproject config .
```

Архітектура проєкту побудована за модульним принципом декомпозиції. Базова структура директорій розробленого монолітного додатка, що ілюструє розділення бізнес-логіки на незалежні компоненти, має вигляд, наведений у лістингу 2.1.

Лістинг 2.1 – Структура директорій розробленого проєкту

```
corporate_portal/
├── config/          # Головна директорія конфігурацій ядра (settings.py, urls.py)
├── users/           # Модуль управління профілями та автентифікацією
├── tasks/           # Підсистема трекінгу інфраструктурних завдань
├── vault/           # Підсистема криптографічного зберігання секретів
├── network/         # Інструментарій мережевої діагностики
├── templates/       # Глобальні HTML-шаблони користувацького інтерфейсу
└── manage.py        # Системна утиліта управління локальним сервером
```

Для інтеграції створених підсистем у ядро фреймворку та визначення кастомної моделі користувача як базової для автентифікації, до центрального конфігураційного файлу `settings.py` було внесено відповідні налаштування (лістинг 2.2).

Лістинг 2.2 – Реєстрація модулів та моделі користувача (файл `settings.py`)

```
INSTALLED_APPS: list[str] = [
    "django.contrib.admin",
```

```

"django.contrib.auth",
"django.contrib.contenttypes",
"django.contrib.sessions",
"django.contrib.messages",
"django.contrib.staticfiles",
"users",
"tasks",
"vault",
"network",
]
AUTH_USER_MODEL: str = "users.CustomUser"

```

Етап 3. Налаштування взаємодії з PostgreSQL.

За замовчуванням під час ініціалізації проєкту фреймворк Django налаштований на використання вбудованої СКБД SQLite, яка зберігає всі дані локально в одному файлі всередині директорії проєкту. Проте в межах розробки корпоративного сайту телекомунікаційного підприємства використання SQLite є неефективним і недоцільним з інженерної точки зору. З метою забезпечення високої транзакційної стійкості, безпеки та масштабованості системи, на етапі проєктування було обґрунтовано вибір об'єктно-реляційної системи керування базами даних PostgreSQL.

У порівнянні з SQLite, яка орієнтована на вбудовані рішення та локальне тестування, СКБД PostgreSQL має низку фундаментальних архітектурних переваг, що є критично важливими для даної кваліфікаційної роботи:

1. **Модель конкурентного доступу.** SQLite зберігає базу в локальному файлі та координує паралельні операції за допомогою файлових блокувань; одночасні читання підтримуються, однак запис потребує ексклюзивного режиму на визначеному етапі транзакції [19]. Для багатокористувацького корпоративного сайту обрано PostgreSQL із клієнт-серверною архітектурою та механізмом MVCC, що краще відповідає сценарію одночасного оновлення завдань, вузлів і секретів кількома працівниками [20; 21].
2. **Підтримка спеціалізованих типів.** PostgreSQL містить типи `inet` і `cidr` для зберігання IP-адрес та мереж, а також тип `jsonb` для структурованих даних із можливістю індексування [22]. Ці можливості є релевантними

для подальшого розвитку модуля Network і зберігання параметрів обладнання.

3. Транзакційність і відновлення. Перевага PostgreSQL у межах проєкту полягає у відповідності клієнт-серверній моделі розгортання, розвинених механізмах конкурентного доступу, адміністрування та масштабування.

Для інтеграції PostgreSQL у середовище Django в ізольованому віртуальному оточенні було встановлено адаптер `psycopg2-binary`. З метою дотримання передових індустріальних практик безпечної розробки (Secure SDLC), підключення до СКБД PostgreSQL реалізовано з використанням змінних середовища операційної системи. Використання системної бібліотеки `os` (лістинг 2.3) унеможливорює компрометацію облікових даних бази даних у разі витоку вихідного коду.

Лістинг 2.3 – Конфігурація підключення до СКБД (файл `settings.py`)

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.environ["POSTGRES_DB"],
        "USER": os.environ["POSTGRES_USER"],
        "PASSWORD": os.environ["POSTGRES_PASSWORD"],
        "HOST": os.environ["POSTGRES_HOST"],
        "PORT": os.environ["POSTGRES_PORT"],
    }
}
```

Завершальним кроком ініціалізації стала генерація файлів міграцій та їх застосування до PostgreSQL за допомогою команд *makemigrations* та *migrate*. Це дозволило Django ORM (Object-Relational Mapping) автоматично перетворити Python-класи моделей на SQL-таблиці без написання "сирих" SQL-запитів, що додатково захищає систему від атак типу SQL Injection на базовому рівні.

2.3 Програмна реалізація модуля управління завданнями

Модуль управління завданнями (Task Manager) є одним із базових функціональних компонентів розроблюваного корпоративного сайту. Його

призначення полягає в автоматизації внутрішніх операційних процесів підприємства: розподілі задач між інженерами мережі, моніторингу поточного стану виконання робіт та забезпеченні взаємодії між менеджментом і технічним персоналом.

Реалізація модуля підпорядкована архітектурному шаблону MVT, де кожен рівень виконує ізольовану функцію, взаємодіючи з іншими через стандартизовані інтерфейси фреймворку Django:

1. **Model:** Визначає структуру даних у СУБД PostgreSQL та правила валідації через Django ORM.
2. **View:** Організовує бізнес-логіку обробки HTTP-запитів, взаємодіє з моделями для вибірки/модифікації даних та передає контекст у шаблони.
3. **Template:** Відповідає за фінальний рендеринг користувацького інтерфейсу на основі HTML та фреймворку інструментів Bootstrap 5.

2.3.1 Проєктування та програмна реалізація моделей даних

Компонент *Model* у межах додатка *tasks* представлений класом *Task*, що успадковує функціонал базового класу *models.Model*. Для забезпечення гнучкості та відповідності бізнес-логіці підприємства, стан виконання завдання типізовано за допомогою вбудованого класу *models.TextChoices*. Це дозволяє уникнути використання magic strings у коді та жорстко обмежити варіанти статусу завдання на рівні системи.

У лістингу 2.4 наведено програмну реалізацію структури даних додатка *tasks*, яка описує сутність завдання та її зв'язки з кастомною моделлю користувача.

Лістинг 2.4 – Програмна реалізація моделі даних *Task* (файл *tasks/models.py*)

```
from __future__ import annotations

from typing import TYPE_CHECKING

from django.conf import settings
from django.db import models
```

```

if TYPE_CHECKING:
    from users.models import CustomUser

class TaskStatus(models.TextChoices):
    TODO = "TODO", "To Do"
    IN_PROGRESS = "IN_PROGRESS", "In Progress"
    DONE = "DONE", "Done"

class TaskPriority(models.TextChoices):
    LOW = "LOW", "Low"
    MEDIUM = "MEDIUM", "Medium"
    HIGH = "HIGH", "High"
    URGENT = "URGENT", "Urgent"
class TaskQuerySet(models.QuerySet["Task"]):
    def with_related(self) -> "TaskQuerySet":
        return self.select_related("created_by", "assigned_to")

TaskManager = models.Manager.from_queryset(TaskQuerySet)

class Task(models.Model):
    title = models.CharField(max_length=255)
    description = models.TextField()

    status = models.CharField(
        max_length=20,
        choices=TaskStatus.choices,
        default=TaskStatus.TODO,
        db_index=True,
    )

    priority = models.CharField(
        max_length=20,
        choices=TaskPriority.choices,
        default=TaskPriority.MEDIUM,
        db_index=True,
    )

    deadline = models.DateField(
        null=True,
        blank=True,
        verbose_name="Дедлайн",
    )

    created_by: CustomUser
    created_by = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        related_name="created_tasks",

```

```

        on_delete=models.SET_NULL,
    )

    assigned_to: CustomUser
    assigned_to = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        related_name="assigned_tasks",
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
    )

    created_at: models.DateTimeField
    created_at = models.DateTimeField(auto_now_add=True)

    updated_at: models.DateTimeField
    updated_at = models.DateTimeField(auto_now=True)

    objects = TaskManager()

    class Meta:
        ordering = ["-created_at"]
        verbose_name = "Task"
        verbose_name_plural = "Tasks"

    def __str__(self) -> str:
        return f'{self.title} ({self.get_status_display()})'

```

2.3.2 Аналіз інженерних рішень та цілісності даних моделі

Під час проєктування моделі Task було закладено кілька критично важливих інженерних рішень, що безпосередньо впливають на реляційну цілісність даних у СУБД PostgreSQL у внутрішній архітектурі проєкту:

1. **Динамічний зв'язок з моделлю користувача:** Замість прямого імпорту класу *CustomUser*, у полі *ForeignKey* використано конфігураційний параметр *settings.AUTH_USER_MODEL*. Це унеможливлює виникнення циклічних імпортів між додатками *users* та *tasks* на етапі ініціалізації ядра Django.
2. **Оптимізація стратегій видалення сутностей (on_delete):**
 - Для поля *created_by* встановлено стратегію *models.CASCADE*. Це означає, що у разі видалення профілю менеджера, всі створені ним

завдання будуть автоматично видалені з бази даних, запобігаючи появі "завдань-сиріт".

- Для поля *assigned_to* застосовано більш гнучку стратегію *models.SET_NULL* у поєднанні з параметрами *null=True* та *blank=True*. У структурі компанії недрпустимо, щоб плинність кадрів або перерозподіл інженерів призводили до втрати історії виконання технічних завдань. Якщо обліковий запис інженера видаляється із системи, саме завдання залишається в базі даних, а поле виконавця просто набуває значення *NULL*, що сигналізує системі про необхідність призначення нового співробітника.

3. **Параметри метаданих:** Опис класу *Meta* дозволяє перевизначити поведінку Django ORM на рівні генерації SQL-запитів. Параметр *ordering = ['-created_at']* інструктує СКБД PostgreSQL за замовчуванням застосовувати сортування за спаданням дати створення (*ORDER BY created_at DESC*).

2.3.3 Реалізація бізнес-логіки представлень та інтерфейсу користувача модуля завдань

Компоненти *View* та *Template* у межах додатка управління завданнями забезпечують обробку вхідних HTTP-запитів, взаємодію з базою даних через Django ORM та формування безпечного графічного інтерфейсу. Для реалізації бізнес-логіки було використано високорівневі представлення на основі класів, що дозволяє інкапсулювати складну логіку фільтрації та перевірки прав доступу.

У лістингу 2.5 наведено реалізацію контролерів формування переліку завдань і перевірки об'єктних прав доступу. Метод *get_queryset()* виконує фільтрацію засобами Django ORM, а функція *can_manage_task()* централізує політику авторизації. Отримання об'єкта через *get_object_or_404()* забезпечує коректне повернення статусу HTTP 404, якщо завдання з указаним ідентифікатором відсутнє.

Лістинг 2.5 – Реалізація контролерів фільтрації завдань і перевірки прав доступу (файл tasks/views.py)

```

from typing import Any

from django.contrib.auth.mixins import LoginRequiredMixin
from django.db.models import QuerySet
from django.http import HttpRequest, HttpResponse
from django.shortcuts import get_object_or_404, render
from django.urls import reverse_lazy
from django.utils import timezone
from django.views.generic import ListView, UpdateView

from tasks.forms import TaskForm
from tasks.models import Task, TaskStatus
from users.models import CustomUser

def can_manage_task(user: CustomUser, task: Task) -> bool:
    return (
        task.created_by_id == user.id
        or user.role in ["ADMIN", "MANAGER"]
    )

class TaskListView(LoginRequiredMixin, ListView):
    model = Task
    template_name = "tasks/task_list.html"
    context_object_name = "tasks"

    def get_queryset(self) -> QuerySet[Task]:
        queryset: QuerySet[Task] = Task.objects.with_related()
        filter_value: str = self.request.GET.get("filter", "all")

        if filter_value == "assigned_to_me":
            queryset = queryset.filter(
                assigned_to=self.request.user
            )
        elif filter_value == "due_today":
            queryset = queryset.filter(
                deadline=timezone.localtime()
            )
        elif filter_value == "todo":
            queryset = queryset.filter(
                status=TaskStatus.TODO
            )
        elif filter_value == "in_progress":
            queryset = queryset.filter(
                status=TaskStatus.IN_PROGRESS
            )
        elif filter_value == "done":

```

```

        queryset = queryset.filter(
            status=TaskStatus.DONE
        )

    return queryset

```

```

class TaskUpdateView(LoginRequiredMixin, UpdateView):
    model = Task
    form_class = TaskForm
    template_name = "tasks/task_form.html"
    context_object_name = "task"
    success_url = reverse_lazy("tasks:task_list")
    object: Task

    def dispatch(
        self,
        request: HttpRequest,
        *args: Any,
        **kwargs: Any,
    ) -> HttpResponse:
        pk: int = int(kwargs["pk"])
        task: Task = get_object_or_404(
            Task.objects.with_related(),
            pk=pk,
        )

        if not can_manage_task(request.user, task):
            return render(
                request,
                "tasks/permission_denied.html",
                status=403,
            )

        self.object = task
        return super().dispatch(request, *args, **kwargs)

    def get_object(
        self,
        queryset: QuerySet[Task] | None = None,
    ) -> Task:
        if hasattr(self, "object"):
            return self.object

        pk: int = int(self.kwargs["pk"])
        return get_object_or_404(
            Task.objects.with_related(),
            pk=pk,
        )

```

Аналіз архітектурних рішень та механізмів безпеки, реалізованих у представленнях та шаблонах:

Рольова модель управління доступом (RBAC) та ізоляція об'єктів: У корпоративному середовищі критично важливо розмежувати повноваження персоналу. У методі *dispatch* класу *TaskUpdateView* реалізовано жорстку перевірку (Object-Level Permissions). Перегляд завдань (*TaskDetailView*) залишено відкритим для всього авторизованого персоналу для забезпечення прозорості робочих процесів, проте функція модифікації або видалення доступна виключно автору завдання або користувачам з привілейованими ролями ADMIN та MANAGER. У разі несанкціонованого доступу система перериває виконання запиту та повертає код помилки 403 Forbidden разом із відповідним графічним попередженням.

Для формалізації політик управління доступом у підсистемі завдань розроблено таблицю прав доступу (таблиця 2.2), яка жорстко регламентує дозволені операції для кожної з визначених ролей (ADMIN, MANAGER, ENGINEER).

Таблиця 2.2

Таблиця прав доступу ролей (RBAC) для підсистеми завдань

Операція системи	Роль: ADMIN	Роль: MANAGER	Роль: ENGINEER	Примітка з безпеки
Перегляд списку	Дозволено	Дозволено	Дозволено	Глобальний доступ (Read-only) для прозорості робочих процесів.
Створення	Дозволено	Дозволено	Дозволено	Автор автоматично стає власником об'єкта (created_by).
Редагування	Дозволено (Всі)	Дозволено (Всі)	Обмежено	ENGINEER має право редагувати виключно ті завдання, які він створив.
Видалення	Дозволено (Всі)	Дозволено (Всі)	Обмежено	ENGINEER має право видаляти виключно ті завдання, які він створив.

Динамічна робота з часовими мітками (Deadlines): Клас *TaskListView* містить логіку фільтрації завдань за критерієм терміновості. За допомогою методу *timezone.localdate()* система динамічно обчислює поточну дату сервера та формує оптимізований SQL-запит до СУБД для вибірки лише тих завдань, термін виконання яких спливає в актуальний день (*filter(deadline=today)*). Результат застосування фільтрації завдань за поточною датою наведено на рис. 2.2.

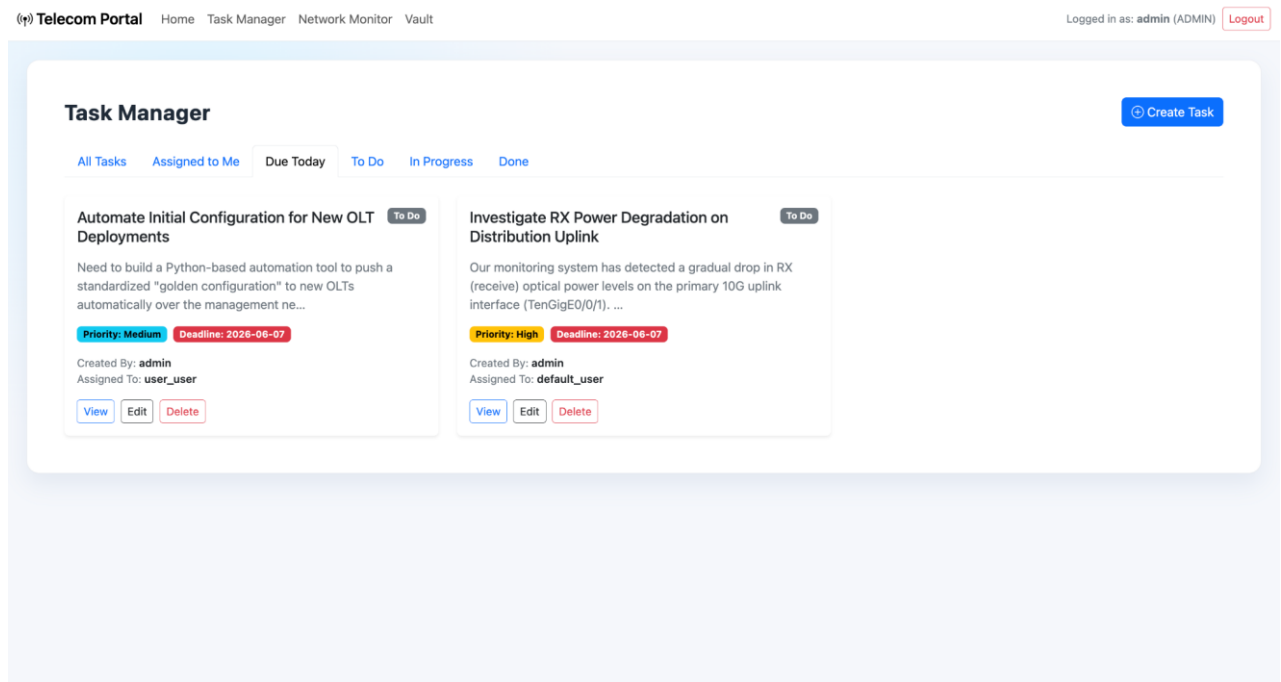


Рис. 2.2 – Відображення фільтрації завдань за призначенням на поточну дату

На рисунку 2.2 також показано і інші реалізовані методи фільтрації (метод *get_queryset()* в лістингу 2.5), такі як:

- Assigned to Me - користувач в цій вкладці бачить тільки ті завдання, які призначені для нього
- Due Today - в цій вкладці зібрані всі завдання, термін виконання (deadline) яких спливає в поточний день перегляду
- To Do/In Progress - всі завдання, із статусом виконання “To Do”/“In Progress”
- “Done” - всі завершені завдання зі статусом “Done”

Безпека на рівні користувацького інтерфейсу (UI Security) імплементована за допомогою умовних конструкцій шаблонізатора. Елементи управління рендеряться виключно у разі позитивної верифікації прав поточного сеансу (лістинг 2.6). Приховування елементів керування на рівні шаблонів зменшує ймовірність помилкових дій користувача, однак не розглядається як самостійний механізм безпеки. Основний контроль доступу реалізовано на серверному рівні в методі *dispatch()*, який перевіряє роль користувача та його зв'язок із конкретним об'єктом.

Лістинг 2.6 – Механізм приховування привілейованих елементів інтерфейсу
(файл *task_list.html*)

```
{% if task.created_by == request.user or request.user.role == 'ADMIN' or request.user.role == 'MANAGER' %}
  <a href="{% url 'tasks:task_edit' task.pk %}" class="btn btn-sm btn-outline-dark">Edit</a>
  <form action="{% url 'tasks:task_delete' task.pk %}" method="post" class="d-inline">
    {% csrf_token %}
    <button type="submit" class="btn btn-sm btn-outline-danger" onclick="return confirm('Are you sure you want to delete this task?');">Delete</button>
  </form>
{% endif %}
```

На рисунку 2.3 наведено форму створення нового інфраструктурного завдання, а на рисунку 2.4 – загальний вигляд списку завдань після їх ініціалізації в системі для користувача ролі ENGINEER.

Telecom Portal

HomeTask ManagerNetwork MonitorVault

Logged in as: admin (ADMIN)Logout

Create Task

Back to List

Title

Description

Status

To Do

Priority

Medium

Deadline

ДД.ММ.ГГГГ

Assigned To

Save Task

Рис. 2.3 – Інтерфейс форми ініціалізації інфраструктурного завдання

Telecom Portal

HomeTask ManagerNetwork MonitorVault

Logged in as: default_user (ENGINEER)Logout

Task Manager

Create Task

All TasksAssigned to MeDue TodayTo DoIn ProgressDone

Vault Module

In Progress

Create Vault Module to Telecom Portal

Priority: MediumDeadline: 2026-06-16

Created By: managerAssigned To: default_user

View

Extra Call

To Do

Quick call with some updates

Priority: UrgentDeadline: 2026-06-04

Created By: managerAssigned To: default_user

View

Plan Weekly Brief

To Do

Plan extra weekly brief with developers

Priority: HighDeadline: 2026-06-04

Created By: adminAssigned To: manager

View

Fix in network/views.py

To Do

Edit the security alert

Priority: MediumDeadline: 2026-06-10

Created By: adminAssigned To: default_user

View

Рис. 2.4 – Відображення переліку завдань для користувача ролі ENGINEER

На рисунку 2.5 відображено загальний вигляд списку завдань для користувачів ролі ADMIN, що виглядає ідентично для користувачів ролі MANAGER

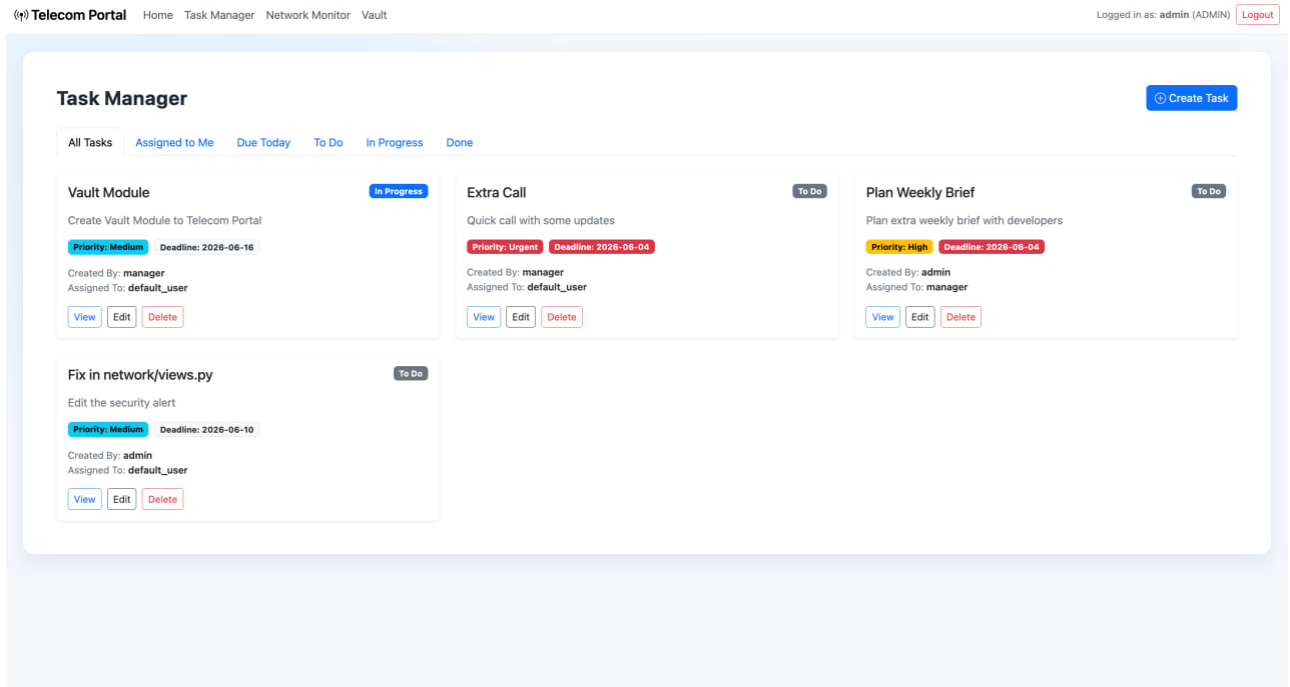


Рис. 2.5 – Відображення переліку завдань для користувача ролі ADMIN та MANAGER

Порівняння інтерфейсів підтверджує, що для користувача з роллю ENGINEER елементи редагування та видалення недоступні за відсутності відповідних повноважень. У разі спроби прямого звернення до контролера редагування або видалення, шляхом підміни URL-адреси:

/tasks/17/edit/

користувачем, який не має відповідних повноважень згідно з таблицею, система перериває обробку запиту та повертає статус HTTP 403 (рис. 2.6).

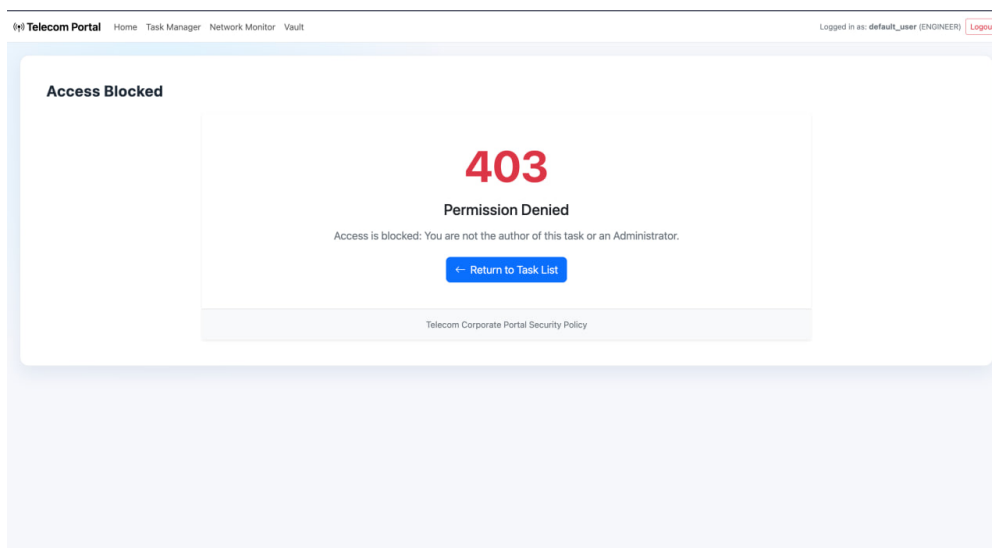


Рис. 2.6 – Блокування несанкціонованого доступу до операції редагування (HTTP 403 Forbidden)

Захист від деструктивних дій: Для критичних операцій, таких як видалення інфраструктурного завдання, на фронтенд-частині імплементовано механізм клієнтського підтвердження (JavaScript Confirm Dialog). Це мінімізує ризики випадкової втрати даних через людський фактор, наприклад, помилкове натискання на кнопку видалення.

Лістинг 2.7 – Програмна реалізація механізму клієнтського підтвердження

```
<form action="{% url 'tasks:task_delete' task.pk %}" method="post" class="d-inline">
  {% csrf_token %}
  <button type="submit" class="btn btn-sm btn-outline-danger" onclick="return confirm('Are you
sure you want to delete this task?');">Delete</button>
</form>
```

2.4 Програмна реалізація підсистеми моніторингу мережевих вузлів

Підсистема моніторингу мережі (Network Monitor) розроблена як спеціалізований інструментарій для інженерів центру управління мережею NOC. Її головне завдання полягає в забезпеченні оперативного контролю доступності корпоративного обладнання та діагностики магістральних каналів зв'язку за допомогою базових мережевих протоколів.

2.4.1 Структура даних інвентаризації мережевих активів

Для ведення обліку телекомунікаційного обладнання провайдера (маршрутизаторів, комутаторів агрегації та доступу) у базі даних PostgreSQL було спроектовано модель `NetworkNode`. Модель містить поля для збереження логічного імені пристрою (`hostname`) та його мережевої адреси (`ip_address`), що виконує роль системи інвентаризації активів.

Використання цієї моделі на рівні інтерфейсу дозволяє технічному персоналу оперувати зрозумілими назвами вузлів замість запам'ятовування їхніх числових IP-адрес, що суттєво знижує ймовірність помилок під час кризових ситуацій у мережі.

Лістинг 2.8 – Програмна реалізація моделі даних `NetworkNode` (файл `network/models.py`)

```
from __future__ import annotations
from django.db import models

class NetworkNode(models.Model):
    hostname = models.CharField(max_length=255, unique=True)
    ip_address = models.GenericIPAddressField(protocol="both", unpack_ipv4=False,
unique=True)
    status = models.CharField(max_length=50, default="UNKNOWN")

    created_at: models.DateTimeField
    created_at = models.DateTimeField(auto_now_add=True)

    updated_at: models.DateTimeField
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering: list[str] = ["hostname"]
        verbose_name = "Network Node"
        verbose_name_plural = "Network Nodes"

    def __str__(self) -> str:
        return f"{self.hostname} ({self.ip_address}) - {self.status}"
```

2.4.2 Реалізація діагностичного інструментарію

Модуль Network використовує системні утиліти ping і traceroute як зовнішні засоби отримання діагностичних даних.

- *ping* надсилає ICMP Echo Request і аналізує Echo Reply, що дає змогу оцінити доступність вузла, затримку передавання та втрати пакетів [23; 24]. У реалізованому представленні кількість запитів обмежено параметром -c 3, тому операція має прогнозовану тривалість і не створює безперервного процесу.
- *traceroute* визначає послідовність проміжних вузлів шляхом надсилання пакетів зі зростаючим значенням TTL. Маршрутизатор, на якому TTL зменшується до нуля, повертає ICMP Time Exceeded, що дає змогу відновити маршрут до цільового вузла [23; 24]. Параметр -n вимикає перетворення IP-адрес на домкнні імена; у межах розробленого модуля це зменшує залежність часу виконання від DNS-відповідей.

Інтеграція зазначених утиліт у середовище фреймворку Django реалізована за допомогою системної бібліотеки Python subprocess. У лістингу 2.9 наведено фрагмент контролера *NetworkDiagnosticView*, який відповідає за формування та виконання системних команд.

Лістинг 2.9 – Початкова вразлива програмна реалізація обробки мережесих команд (файл network/views.py)

```
if selected_tool == "traceroute":
    command: str = f"traceroute -n {target}"
else:
    command: str = f"ping -c 3 {target}"
try:
    result: subprocess.CompletedProcess[str] = subprocess.run(
        command,
        shell=True,
        capture_output=True,
        text=True,
        timeout=25,
        check=False,
    )
    output = (result.stdout or "") + (result.stderr or "")
except subprocess.TimeoutExpired:
```

output = "Diagnostic timeout: the command exceeded 25 seconds."

Важливою архітектурною особливістю наведеної реалізації є передача сформованої команди у вигляді суцільного рядка та використання аргументу `shell=True` під час виклику методу `subprocess.run()`. З інженерної точки зору, такий підхід суттєво спрощує маршрутизацію виводу та взаємодію з системними змінними середовища.

Однак, з позиції інформаційної безпеки, делегування парсингу метасимволів безпосередньо командній оболонці операційної системи створює класичний вектор атаки впровадження команд (CWE-78). Зважаючи на дослідницький характер кваліфікаційної роботи, дана специфіка імплементації була залишена усвідомлено як контрольована архітектурна вразливість. Вона виступатиме об'єктом практичної експлуатації та наступної рефакторизації під час проведення практичного дослідження безпеки у третьому розділі даної роботи.

На рис. 2.7 та рис. 2.8 продемонстровано результати успішного виконання мережевих тестів до заданих цільових вузлів через інтерфейс корпоративного порталу.

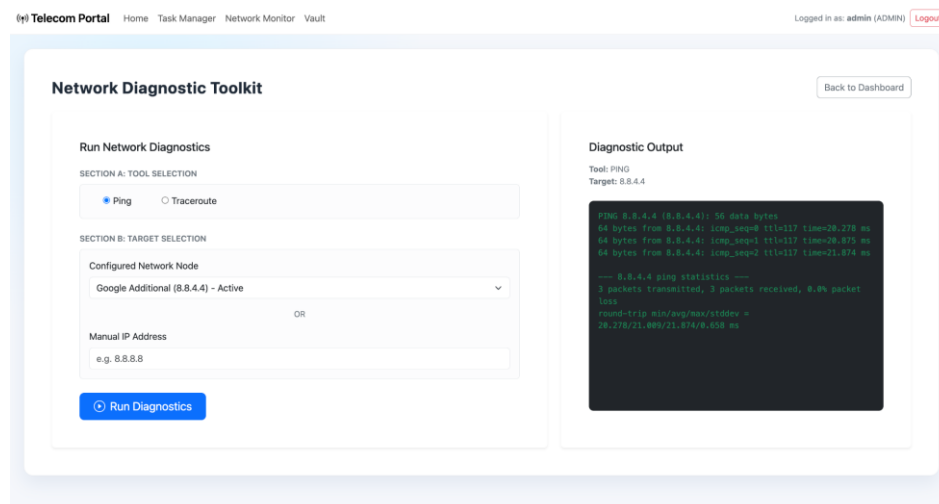


Рис. 2.7 – Результат виконання ICMP-діагностики (ping) цільового вузла

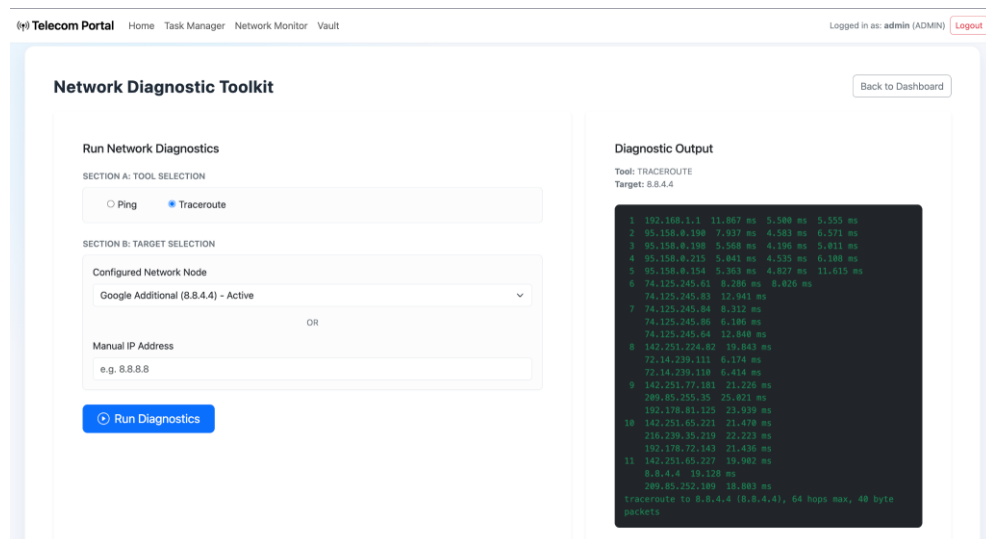


Рис. 2.8 – Маршрутизація пакетів (traceroute) без застосування DNS-резолвінгу

2.4.3 Аналіз архітектурних рішень підсистеми

Розроблена конфігурація модуля моніторингу спирається на наступні інженерні рішення:

1. **Гнучкість маршрутизації параметрів:** Логіка контролера побудована таким чином, що пріоритет віддається об'єктам із бази даних (*node.ip_address*) – поле Configured Network Node на рисунку 2.7, однак у разі необхідності діагностики зовнішнього ресурсу інженер може використати ручне введення цільової адреси (*custom_ip*) – поле manual IP Address на рисунку 2.7.
2. **Організація відмовостійкості процесу:** Використання конструкції *try...except* та параметра *timeout=25* у функції *subprocess.run()* гарантує, що у разі відсутності відповіді від цільового вузла, наприклад, через фільтрацію ICMP-трафіку брандмауером, процес утиліти буде примусово завершено операційною системою. Це запобігає виснаженню пулу потоків веб-сервера (Thread Pool Exhaustion) під час блокувальних мережеских викликів.
3. **Пряма взаємодія з системною оболонкою (Shell Integration):** У методі запуску процесу використано аргумент *shell=True*, а команда передається як єдиний форматований рядок (*f'ping -c 3 {target}'*). З архітектурної

точки зору це забезпечує найвищий рівень сумісності з різними дистрибутивами Unix-подібних операційних систем, дозволяючи інтерпретатору командного рядка самостійно керувати змінними середовища та конвеєрами. Однак, такий підхід висуває критичні вимоги до надійності механізмів валідації вхідних даних на рівні форм. Оцінка стійкості реалізованого підходу до кібератак класу Command Injection є предметом детального аналізу у наступному розділі роботи.

2.5 Програмна реалізація підсистеми безпечного зберігання інфраструктурних конфігурацій (Vault)

Підсистема Vault призначена для зберігання паролів до мережевого обладнання, ключів протоколів, SNMP-рядків спільноти та інших інфраструктурних секретів. Збереження таких значень у відкритому вигляді створює ризик їх розкриття через резервну копію, дамп бази даних або помилково наданий доступ. Тому для полів із секретними даними застосовано шифрування на рівні застосунку, а доступ до об'єктів додатково обмежено рольовими та об'єктними правилами [4; 5].

2.5.1 Архітектура криптографічної моделі та управління даними

Для криптографічної обробки використано клас Fernet бібліотеки cryptography. Формат Fernet поєднує шифрування AES із 128-бітним ключем у режимі CBC, доповнення PKCS7 і HMAC-SHA256 для перевірки автентичності повідомлення [14]. Ключ Fernet є URL-safe base64-поданням 32 байтів ключового матеріалу й повинен зберігатися окремо від шифротекстів та вихідного коду. Отже, безпека модуля визначається двома незалежними умовами: коректним використанням бібліотечної реалізації та захищеним керуванням ключем [5; 14; 25].

У межах компонента Model (файл vault/models.py) спроектовано реляційну сутність Secret, структуру якої наведено у лістингу 2.10.

Лістинг 2.10 – Структура реляційної моделі зберігання секретів (файл vault/models.py)

```
class Secret(models.Model):
    ACCESS_CHOICES: list[tuple[str, str]] = [
        ("everyone", "Everyone"),
        ("role", "Specific Role"),
        ("user", "Single User"),
    ]
    name = models.CharField(max_length=255, unique=True)
    encrypted_value = models.TextField(verbose_name="Encrypted Value", blank=True)
    description = models.TextField(verbose_name="Description", blank=True, null=True)

    access_type = models.CharField(max_length=10, choices=ACCESS_CHOICES,
default="everyone")
    assigned_role = models.CharField(
        max_length=20, blank=True, null=True, help_text="e.g., ADMIN, MANAGER, ENGINEER"
    )
    assigned_user = models.ForeignKey(
        settings.AUTH_USER_MODEL, on_delete=models.SET_NULL,
        null=True, blank=True, related_name="assigned_secrets"
    )
    created_by = models.ForeignKey(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="vault_secrets"
    )
    created_at: models.DateTimeField
    created_at = models.DateTimeField(auto_now_add=True)
```

Для перехоплення та конвертації відкритого тексту перед транзакцією у СУБД реалізовано методи інкапсуляції:

- `set_secret()`: Відповідає за перетворення вхідного пароля (UTF-8) на набір байтів, його шифрування алгоритмом AES та фінальне декодування токена у безпечний текстовий формат Base64 для збереження у полі типу `TextField`.
- `get_secret()`: Виконує зворотну операцію дешифрування під час авторизованого сеансу користувача. Для забезпечення надійності роботи системи в методі також імплементовано механізм зворотної сумісності (Backward Compatibility), який коректно обробляє записи, що могли бути історично збережені у форматі сирих байтів (рядок `b'...'`).

Реалізацію зазначених методів наведено у лістингу 2.11.

Лістинг 2.11 – Програмна реалізація методів шифрування та дешифрування даних

```
def set_secret(self, raw_password: str) -> None:
    secret_bytes: bytes = raw_password.encode("utf-8")
    token: bytes = FERNET_CIPHER.encrypt(secret_bytes)
    self.encrypted_value = token.decode("utf-8")

def get_secret(self) -> str:
    raw_value: str = (self.encrypted_value or "").strip()
    if not raw_value:
        return ""

    token_bytes: bytes
    if raw_value.startswith("b") or raw_value.startswith('b'):
        # Механізм зворотної сумісності для історичних записів
        parsed: object = ast.literal_eval(raw_value)
        if not isinstance(parsed, bytes):
            raise InvalidToken("Stored token is not valid bytes.")
        token_bytes = parsed
    else:
        token_bytes = raw_value.encode("utf-8")
    decrypted: bytes = FERNET_CIPHER.decrypt(token_bytes)
    return decrypted.decode("utf-8")
```

Ефективність застосованого підходу до шифрування наочно підтверджується під час інспектування записів безпосередньо через адміністративну панель бази даних. На рисунку 2.9 продемонстровано сутність збереженого інфраструктурного об'єкта. Навіть за умови отримання зловмисником повного несанкціонованого доступу до СУБД, наприклад, шляхом експлуатації SQL-ін'єкції або викраденням резервної копії, скомпрометований пароль матиме вигляд криптографічного токена. Дешифрування цього масиву даних є обчислювально нездійсненним за умови збереження секретності ключа.

The screenshot shows a 'Change Secret' form. At the top, there's a 'SECRET' label and a 'HISTORY' button. The form contains the following fields:

- Name:** SECRET
- Description:** ADMINS ONLY
- Encrypted Value:** gAAAAABqHUBtXgGqur1zdbJmtHkhE82lqBCMEXbeYRHCP7TLtSulCJaGEkTOrkYcjHwebIbPH8xXbVRJcOdXdx8rKIETNS1A==
- Created by:** admin
- Created at:** June 1, 2026, 11:18 a.m.

At the bottom, there are four buttons: 'SAVE', 'Save and add another', 'Save and continue editing', and 'Delete'.

Рис. 2.9 – Відображення зашифрованого криптографічного токена у базі даних, ілюстрація концепції Data at Rest

Важливою архітектурною особливістю поточної ітерації розробки є інтеграція криптографічного ключа (*FERNET_KEY*) безпосередньо у вихідний код моделі. З точки зору безпечного управління ключами це класифікується як використання жорстко закодованих облікових даних (CWE-321: Use of Hard-coded Cryptographic Key). Дане інженерне рішення було імплементовано усвідомлено як контрольовану слабкість системи, що виступатиме об'єктом дослідження під час проведення практичного дослідження безпеки у третьому розділі роботи.

2.5.2 Гранулярний контроль доступу та логіка представлень

Для мінімізації впливу внутрішніх загроз у модулі імплементовано гнучку рольову модель контролю доступу на рівні конкретних записів. Поле *access_type* дозволяє автору секрету визначати стратегію спільного використання інформації. Детальний опис доступних режимів наведено у таблиці 2.3.

Режими доступу до секретів у модулі Vault

Режим доступу (access_type)	Логіка доступу до дешифрування	Цільове використання у телеком-мережі
everyone	Доступно всім авторизованим користувачам системи.	Загальнокорпоративні конфігурації
role	Доступно лише користувачам, чия посада збігається з полем assigned_role.	Паролі до комутаторів рівня доступу
user	Доступно виключно індивідуальному користувачу (assigned_user).	Персональні VPN-сертифікати або унікальні ключі доступу до API.

Перевірка повноважень здійснюється на рівні серверних представлень (файл vault/views.py):

- Ізоляція під час створення (SecretCreateView):** Форма *SecretForm* перехоплює відкритий пароль через віртуальне поле *raw_secret*. У методі *form_valid()* пароль передається до криптографічного методу *set_secret()* ще до збереження об'єкта в базу даних. Як продемонстровано у лістингу 2.12, використання параметра *commit=False* запобігає виконанню операції INSERT до шифрування значення.

Лістинг 2.12 – Перехоплення та шифрування відкритого пароля перед збереженням у БД (файл vault/views.py)

```
class SecretCreateView(LoginRequiredMixin, CreateView):
    model = Secret
    form_class = SecretForm
    template_name = "vault/secret_form.html"
    success_url = reverse_lazy("vault:secret_list")
```

```
def form_valid(self, form):
    # 1. Створення об'єкта в пам'яті без запису в БД
    self.object = form.save(commit=False)
    self.object.created_by = self.request.user

    # 2. Отримання відкритого пароля з форми
    raw_password = form.cleaned_data["raw_secret"]

    # 3. Шифрування пароля
    self.object.set_secret(raw_password)

    # 4. Збереження зашифрованого об'єкта у БД
    self.object.save()

    return HttpResponseRedirect(self.get_success_url())
```

2. Контроль читання (*SecretDetailView*): Логіка методу *get_context_data()* перевіряє права доступу поточного сеансу перед спробою дешифрування токена. У разі відсутності прав або виникнення помилки *InvalidToken*, система перериває дешифрування і генерує безпечне повідомлення (Access Denied) замість аварійного завершення роботи веб сервера (HTTP 500).
3. Обмеження модифікації (*SecretUpdateView* та *SecretDeleteView*): Внесено суворі обмеження на редагування. Перевизначений метод *dispatch()* блокує доступ усім користувачам, окрім автора або адміністратора, повертаючи код HTTP 403 Forbidden.

Програмну імплементацію механізмів контролю доступу під час читання та модифікації криптографічних записів наведено у лістингу 2.13.

Лістинг 2.13 – Серверна перевірка прав доступу в контролерах (файл vault/views.py)

```
class SecretDetailView(LoginRequiredMixin, DetailView):
    # ...
    def get_context_data(self, **kwargs: Any) -> dict[str, Any]:
        context: dict[str, Any] = super().get_context_data(**kwargs)
        secret: Secret = self.object
        user = self.request.user

        # Перевірка прав на дешифрування об'єкта
        has_access: bool = (
            user == secret.created_by
```

```

    or getattr(user, "role", "") == "ADMIN"
    or secret.access_type == "everyone"
    or (secret.access_type == "user" and secret.assigned_user == user)
    or (secret.access_type == "role" and secret.assigned_role == getattr(user, "role", ""))
)
context["access_denied"] = not has_access
# ... подальша логіка дешифрування ...
return context

```

```

class SecretUpdateView(LoginRequiredMixin, UpdateView):
    # ...
    def dispatch(self, request, *args, **kwargs):
        secret: Secret = get_object_or_404(Secret, pk=kwargs["pk"])

        if not (request.user == secret.created_by or getattr(request.user, "role", "") == "ADMIN"):
            return HttpResponseForbidden("Forbidden")
        return super().dispatch(request, *args, **kwargs)

```

На рисунках 2.10–2.12 проілюстровано процес взаємодії користувача з модулем зберігання секретів та роботу механізмів відмови у доступі.

The screenshot shows the 'Add Secret' form in the Telecom Portal. The form has the following elements:

- Name:** A text input field.
- Description:** A larger text input field.
- Access type:** A dropdown menu with three options: 'Everyone' (selected), 'Specific Role', and 'Single User'.
- Assigned user:** A dropdown menu with a placeholder '-----'.
- Секретне значення (Пароль, ключ):** A text input field with the placeholder 'Введіть секрет...'.
- Buttons:** A 'Back to Vault' button in the top right and a 'Save Encrypted Secret' button at the bottom left of the form.

The top navigation bar includes 'Telecom Portal', 'Home', 'Task Manager', 'Network Monitor', and 'Vault'. The user is logged in as 'admin (ADMIN)' with a 'Logout' button.

Рис. 2.10 – Інтерфейс ініціалізації інфраструктурного секрету з вибором політики доступу

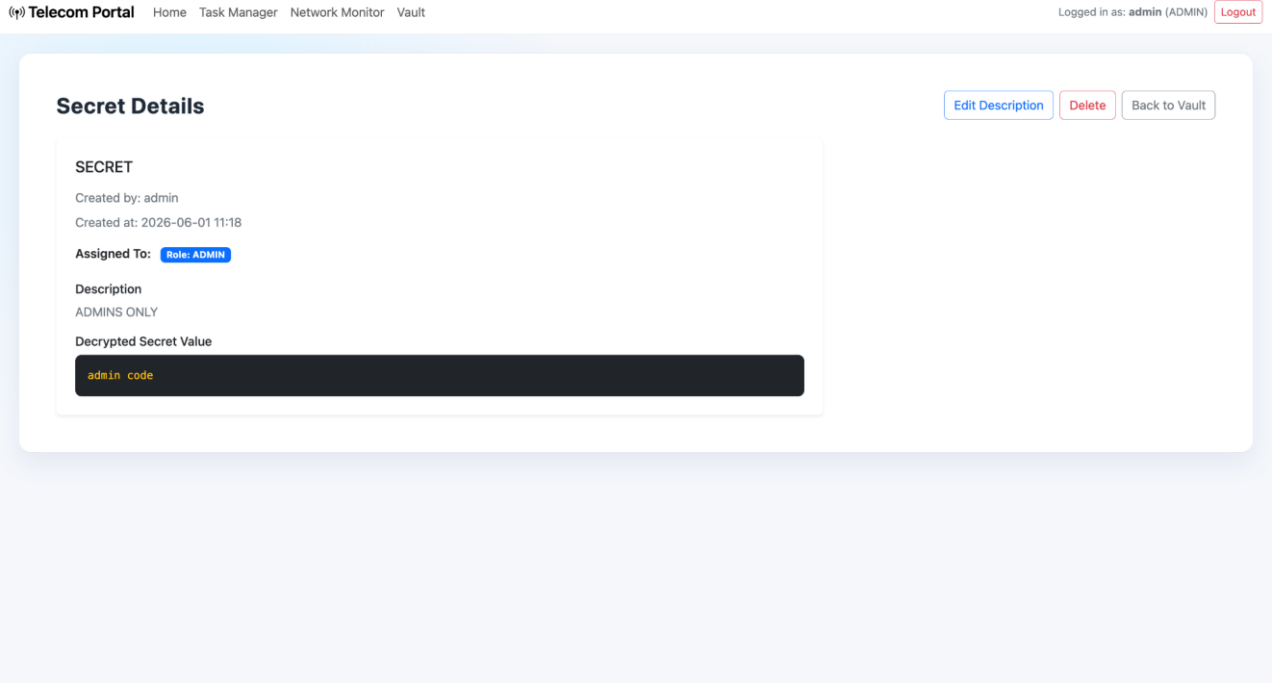


Рис. 2.11 – Успішне дешифрування секрету авторизованим користувачем

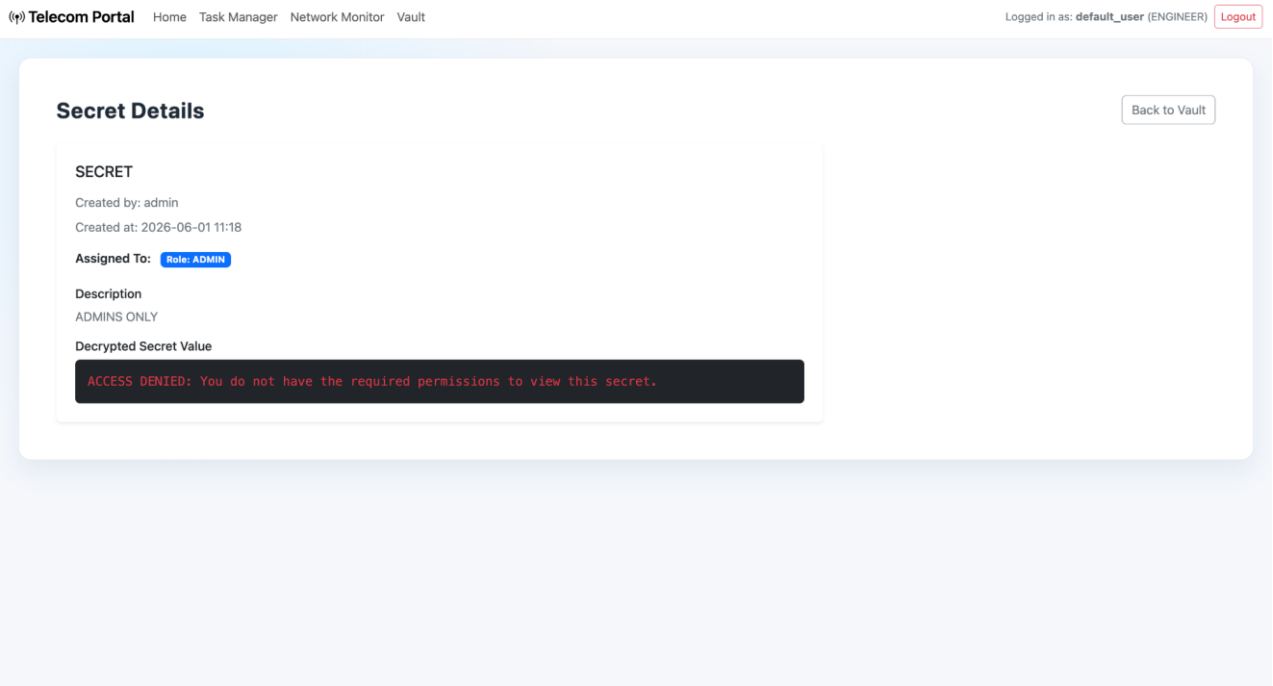


Рис. 2.12 – Блокування доступу до криптографічного об'єкта для користувача із обмеженим доступом

2.5.3 Захист від суперечливих станів на рівні форм

Для забезпечення цілісності реляційних даних розроблено багаторівневий механізм валідації доступу. На фронтенд-частині застосовано динамічний JavaScript, який активує або деактивує селектори вибору ролі чи користувача залежно від обраного типу доступу (*access_type*).

Щоб запобігти маніпуляціям із HTTP-запитами зі сторони клієнта, у класі *SecretForm* (файл *vault/forms.py*) перевизначено метод *clean()*. Серверний механізм, представлений у лістингу 2.14, примусово очищує дані суперечливих полів (наприклад, видаляє ідентифікатор користувача *assigned_user*, якщо обрано рівень доступу *everyone*). Саме за допомогою даного рішення повністю унеможлиблюється виникнення логічних колізій у базі даних.

Лістинг 2.14 – Санітизація суперечливих параметрів доступу у формі (файл *vault/forms.py*)

```
def clean(self) -> dict[str, object]:
    cleaned_data: dict[str, object] = super().clean()
    access_type: str = str(cleaned_data.get("access_type") or "everyone")
    if access_type == "everyone":
        cleaned_data["assigned_role"] = ""
        cleaned_data["assigned_user"] = None
    elif access_type == "role":
        cleaned_data["assigned_user"] = None
    elif access_type == "user":
        cleaned_data["assigned_role"] = ""
    return cleaned_data
```

ВИСНОВКИ

1. У розділі спроектовано архітектуру корпоративного сайту на базі Django 5.2 LTS і PostgreSQL, що забезпечує модульну декомпозицію системи на підсистеми управління завданнями, мережевого моніторингу та захищеного зберігання інфраструктурних секретів.
2. Реалізовано рольову модель доступу та серверні перевірки повноважень на рівні представлень, що дозволяє обмежити модифікацію об'єктів відповідно до ролі користувача та його зв'язку з конкретним записом.

3. Визначено дві контрольовані архітектурні слабкості: використання системної оболонки під час виконання мережових команд і зберігання криптографічного ключа у вихідному коді. Саме ці рішення формують основу для практичного дослідження безпеки в Розділі 3.

РОЗДІЛ 3

АНАЛІЗ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ТА ПРАКТИЧНА ЕКСПЛУАТАЦІЯ ВРАЗЛИВОСТЕЙ

У цьому розділі наведено результати практичного дослідження інформаційної безпеки розробленого корпоративного сайту. Метою аналізу є перевірка того, наскільки реалізовані архітектурні та програмні рішення забезпечують розмежування доступу, безпечне оброблення вхідних даних і захист конфіденційної інформації, що зберігається у підсистемі Vault.

Дослідження проведено методом white-box тестування, який передбачає аналіз вихідного коду та контрольоване відтворення сценаріїв атак. Такий підхід дав змогу визначити причини виявлених недоліків, простежити можливий розвиток атаки та сформувати програмні заходи для зменшення відповідних ризиків.

Об'єктами перевірки обрано три функціональні підсистеми: модуль управління завданнями Tasks, засоби мережевої діагностики Network і сховище інфраструктурних секретів Vault. Перелік тестів сформовано з урахуванням ризиків, визначених у першому розділі: порушення контролю доступу, ін'єкцій команд операційної системи, помилок конфігурації та неналежного керування криптографічними ключами [3; 26].

Дослідження проведено в ізольованому середовищі з використанням тестових облікових записів і синтетичних даних. Перевірки не переобачали впливу на сторонні інформаційні системи або реальне телекомунікаційне обладнання.

3.1 Методика та умови проведення практичного дослідження

Для оцінювання рівня захищеності застосовано поєднання статичного аналізу вихідного коду та динамічного тестування функцій вебдодатка. Статичним аналізом охоплено перевірку представлень Django, форм, моделей даних, шаблонів і параметрів конфігурації. Динамічна частина передбачає

формування контрольованих HTTP-запитів, зміну параметрів URL і введення спеціально підготовлених тестових значень.

Для кожного сценарію визначено передумови, дії користувача, очікувана безпечна реакція системи та фактичний результат. Після формування захисного рішення той самий сценарій розглянуто повторно, що дає змогу порівняти поведінку початкової та вдосконаленої реалізацій.

3.1.1 Модель порушника

У межах дослідження розглянуто внутрішнього порушника, який має чинний обліковий запис із роллю Engineer, але не володіє адміністративними повноваженнями. Така модель відповідає ситуації компрометації облікових даних працівника NOC або спробі перевищення ним наданих прав.

Передбачається, що порушник може:

- використовувати доступні після автентифікації функції сайту;
- змінювати ідентифікатори об'єктів у URL-адресах;
- формувати POST-запити поза штатним інтерфейсом;
- передавати спеціально сформовані значення у поля введення;
- аналізувати відповіді сервера.

Початкового доступу до файлової системи сервера, бази даних, вихідного коду та криптографічного ключа порушник не має. Метою моделювання є визначення того, чи може він отримати такий доступ через помилки прикладної логіки.

3.1.2. Систематизація тестових сценаріїв

Для систематизації аналізу сформовано набір тестових сценаріїв, що охоплює основні ризики, визначені в теоретичній частині роботи. Кожний сценарій пов'язано з конкретним компонентом корпоративного сайту та очікуваною безпечною реакцією системи. Перевірки виконуються на

серверному рівні, оскільки обмеження, реалізовані лише елементами інтерфейсу, можуть бути обійдені шляхом формування прямого HTTP-запиту.

Під час оцінювання результатів використано два критерії. Сценарій вважається успішно пройденим, якщо заборонена дія блокується, стан даних не змінюється, а відповідь сервера не розкриває конфіденційної інформації. Якщо тестове значення спричиняє виконання непередбаченої операції, розкриття даних або обхід установленної політики доступу, результат вважається негативним і потребує виправлення програмної реалізації.

Таблиця 3.1

Таблиця сценаріїв дослідження інформаційної безпеки

Код	Об'єкт перевірки	Тестовий сценарій	Очікувана безпечна реакція
AC-01	Модуль Vault	Користувач із роллю Engineer змінює в URL ідентифікатор та намагається редагувати чужий секрет	Сервер повертає HTTP 403 або 404; дані не змінюються
AC-02	Модуль Vault	Користувач намагається переглянути секрет, до якого йому не надано доступу	Розшифроване значення і конфіденційні метадані не відображаються
AC-03	Модуль Tasks	Користувач намагається змінити або видалити завдання, створене іншим працівником	Операція блокується серверною перевіркою повноважень
VAL-01	Форма SecretForm	У POST-запиті одночасно передаються суперечливі значення access_type, assigned_role та assigned_user	Суперечливі поля очищуються або форма повертає помилку валідації
INJ-01	Модуль Network	У поле адреси передається рядок 8.8.8.8; ls -la	Додаткова команда не виконується; запит відхиляється
SQL-01	Django ORM	У текстове поле передається SQL-подібна конструкція ' OR 1=1 --	Значення обробляється як звичайний текст; структура SQL-запиту не змінюється
XSS-01	Шаблони Tasks	У поле опису завдання передається тестовий HTML/JavaScript-код	Код екранується та відображається як текст без виконання у браузері
CSRF-01	Операції видалення	Надсилається POST-запит без чинного CSRF-токена	Django відхиляє запит зі статусом HTTP 403; об'єкт не видаляється
CFG-01	Конфігурація Django	Виконується команда python manage.py check --deploy	Критичні попередження конфігурації відсутні або документовано усунені
CRY-01	Модуль Vault	Перевіряється наявність ключа Fernet у вихідному коді	Ключ не зберігається безпосередньо у файлах програмного коду
CRY-02	Зашифрований запис Vault	Виконується спроба дешифрування викраденого тестового токена	Без відповідного ключа відновлення початкового значення неможливе

Сценарії AC-01–AC-03 призначені для перевірки об'єктного контролю доступу. Тест VAL-01 оцінює стійкість серверної бізнес-логіки до навмисного формування суперечливих параметрів. Сценарії INJ-01, SQL-01, XSS-01 і CSRF-01 охоплюють безпечність оброблення вхідних даних та HTTP-запитів. Перевірка CFG-01 стосується параметрів розгортання Django, а сценарії CRY-01 і CRY-02 спрямовані на оцінювання захищеності криптографічного ключа та зашифрованих даних.

Сформована таблиця використовується як основа для подальшої фіксації результатів до та після програмного усунення виявлених недоліків. Такий підхід забезпечує послідовне охоплення визначених ризиків і дає змогу оцінити ефективність реалізованих заходів захисту [3; 26].

3.2 Перевірка реалізованих механізмів захисту

На першому етапі практичного дослідження перевірено механізми захисту, реалізовані на рівні представлень, форм, ORM і шаблонів Django. Основну увагу зосереджено на серверному контролі доступу, обробленні суперечливих параметрів, захисті від SQL-ін'єкцій, міжсайтового скриптингу та підроблення міжсайтових запитів.

Перевірки проводилися з позиції автентифікованого користувача з роллю Engineer. Для моделювання несанкціонованих дій використовувалися прямі HTTP-запити та зміна ідентифікаторів об'єктів у URL-адресах. Такий підхід дає змогу перевірити саме серверні обмеження, не покладаючись на приховування кнопок або посилань у користувацькому інтерфейсі.

3.2.1 Перевірка об'єктного контролю доступу

Порушення об'єктного контролю доступу виникає, коли сервер знаходить запитаний об'єкт за його ідентифікатором, але не перевіряє повноваження користувача щодо виконання конкретної операції. У такому разі заміна параметра pk у URL може надати доступ до даних іншого користувача.

Відповідно до OWASP, перевірка прав має виконуватися на серверному рівні під час кожного звернення до захищеного об'єкта [2; 3]. У модулі Tasks право на зміну та видалення завдання визначається функцією *can_manage_task()*. Операція дозволяється автору завдання, адміністратору або менеджеру. Інші користувачі отримують відповідь зі статусом HTTP 403. Відповідний фрагмент програмної реалізації наведено в лістингу 3.1.

Лістинг 3.1 – Серверна перевірка права на управління завданням

```
def can_manage_task(user: CustomUser, task: Task) -> bool:
    return (
        task.created_by_id == user.id
        or user.role in ["ADMIN", "MANAGER"]
    )

class TaskUpdateView(LoginRequiredMixin, UpdateView):
    model = Task
    form_class = TaskForm

    def dispatch(self, request, *args, **kwargs):
        task = get_object_or_404(
            Task.objects.with_related(),
            pk=kwargs["pk"],
        )
        if not can_manage_task(request.user, task):
            return render(
                request,
                "tasks/permission_denied.html",
                status=403,
            )
        self.object = task
        return super().dispatch(request, *args, **kwargs)
```

Використання *get_object_or_404()* забезпечує контрольовану реакцію на звернення до відсутнього запису, а функція *can_manage_task()* централізує правило авторизації. Аналогічна перевірка застосовується у представленні *TaskDeleteView*, тому приховування кнопок редагування та видалення у шаблоні не є єдиним рівнем захисту. Для перевірки сценарію АС-03 користувач Engineer В здійснював пряме звернення до адреси редагування завдання, створеного користувачем Engineer А:

/tasks/5/edit/

За результатами перевірки сервер відхилив запит зі статусом HTTP 403, а вміст завдання залишився незмінним, що наведено на рис. 3.1. Отже, підміна ідентифікатора не дозволила виконати несанкціоноване редагування.

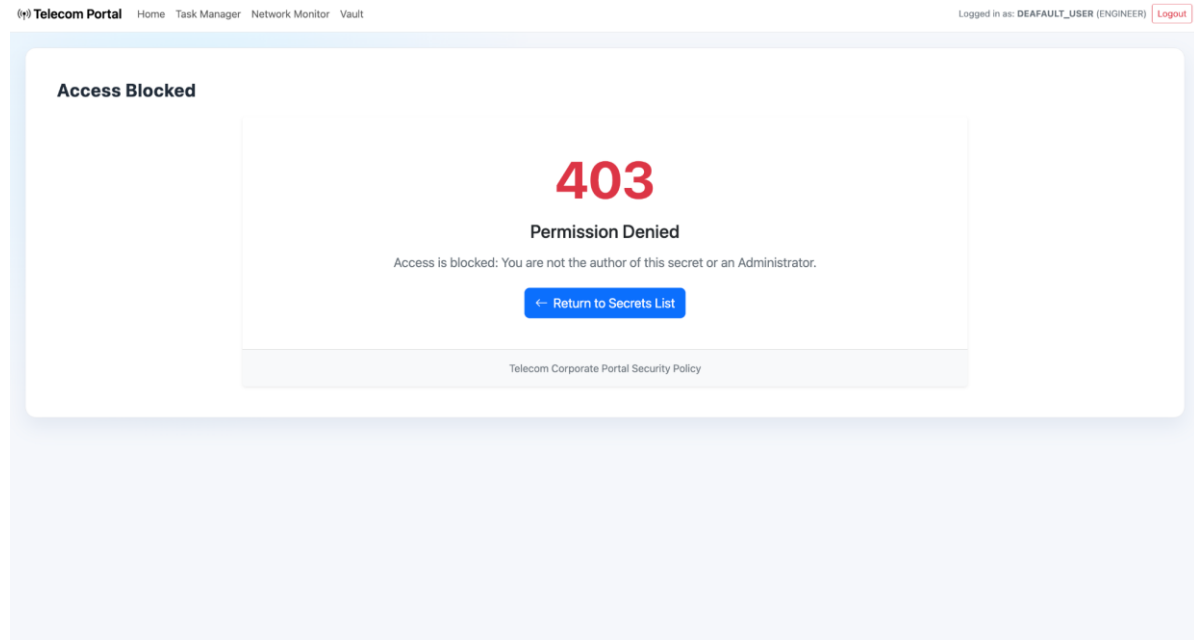


Рис. 3.1 – Блокування несанкціонованого редагування завдання

У модулі Vault серверна авторизація операцій редагування та видалення реалізована безпосередньо в методі `dispatch()`. Доступ надається лише автору секрету або користувачу з роллю Administrator.

Лістинг 3.2 – Перевірка повноважень під час редагування секрету

```
def dispatch(self, request, *args, **kwargs):
    secret = get_object_or_404(
        Secret,
        pk=kwargs["pk"],
    )

    if not (
        request.user == secret.created_by
        or getattr(request.user, "role", "") == "ADMIN"
    ):
        return HttpResponseForbidden("Forbidden")

    return super().dispatch(request, *args, **kwargs)
```

Для сценарію AC-01 користувач Engineer В підставляє у URL ідентифікатор секрету, створеного користувачем Engineer А. Сервер повернув HTTP 403 до виконання операції збереження. Повторна перевірка запису підтвердила, що його опис і параметри доступу не змінилися.

Окремо перевірено перегляд розшифрованого значення секрету. Представлення *SecretDetailView* дозволяє дешифрування лише автору, адміністратору, безпосередньо призначеному користувачу або представнику призначеної ролі. Якщо жодна умова не виконується, метод *get_secret()* не викликається, а в контекст шаблону передається ознака *access_denied*.

Лістинг 3.3 – Перевірка права на дешифрування секретного значення

```
has_access = (
    user == secret.created_by
    or getattr(user, "role", "") == "ADMIN"
    or secret.access_type == "everyone"
    or (
        secret.access_type == "user"
        and secret.assigned_user == user
    )
    or (
        secret.access_type == "role"
        and secret.assigned_role == getattr(user, "role", "")
    )
)

context["access_denied"] = not has_access
context["decrypted_value"] = None

if has_access:
    context["decrypted_value"] = secret.get_secret()
```

За відсутності відповідного права зашифроване значення не передається шаблону у відкритому вигляді. В інтерфейсі користувач отримує повідомлення про відмову в доступі. Результат перевірки наведено на рис. 3.2.

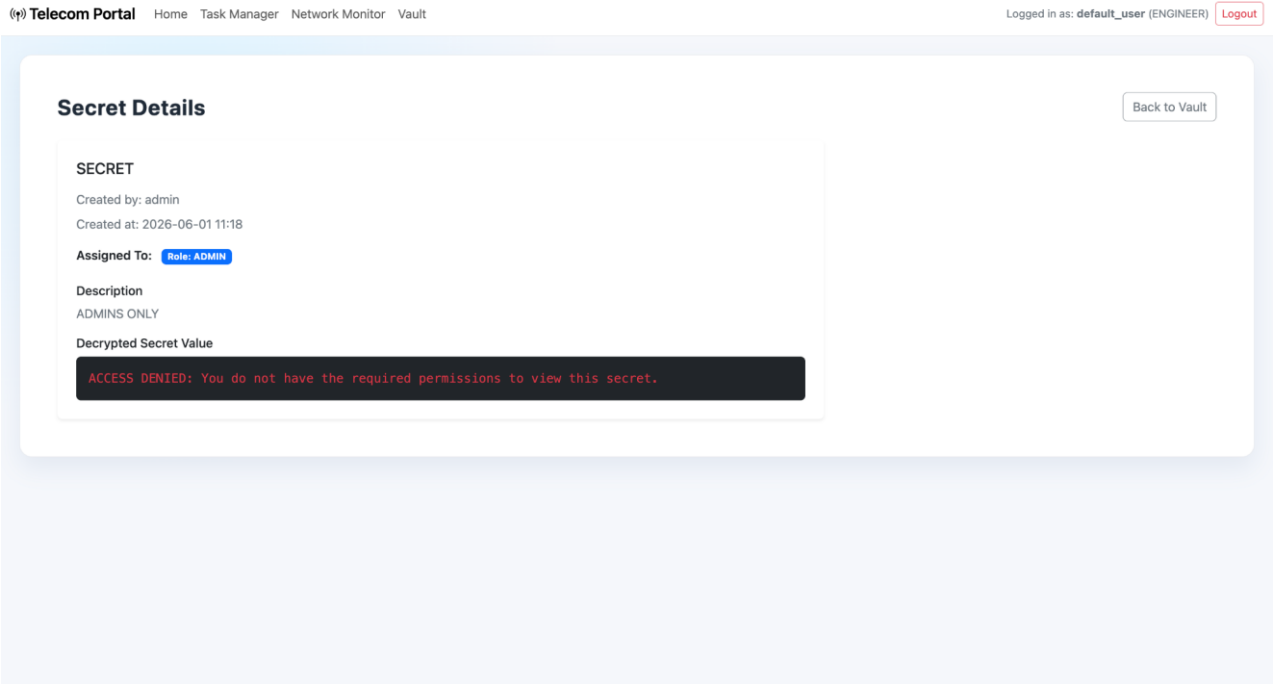


Рис. 3.2 – Блокування перегляду секретного значення користувачем без відповідних повноважень

Результати перевірки наведено в таблиці 3.2.

Таблиця 3.2

Результати перевірки об’єктного контролю доступу

Код	Перевірка	Очікуваний результат	Фактичний результат
АС-01	Редагування чужого секрету	HTTP 403, запис не змінено	Запит заблоковано, запис не змінено
АС-02	Перегляд недоступного секретного значення	Значення не розкривається	Дешифрування не виконується
АС-03	Редагування чужого завдання	HTTP 403, запис не змінено	Запит заблоковано, запис не змінено
АС-04	Видалення чужого завдання	HTTP 403, об’єкт збережено	Запит заблоковано, об’єкт збережено

3.2.2 Перевірка серверної валідації параметрів доступу

Окрім перевірки повноважень користувачів, дослідження охоплювало коректність оброблення взаємозалежних параметрів підсистеми Vault. Під час

створення секрету користувач визначає тип доступу *access_type*, який може набувати значень *everyone*, *role* або *user*. Залежно від вибраного режиму значущим є лише одне з додаткових полів: *assigned_role* або *assigned_user*.

Користувацький інтерфейс приховує або деактивує непотрібні поля, однак цей механізм не можна вважати достатнім засобом захисту. Автентифікований користувач може сформував POST-запит самостійно та передати серверу значення, які неможливо одночасно вибрати через штатну форму. Наприклад, для режиму *everyone* у запиті можуть додатково міститися ідентифікатор користувача та назва ролі. Такий сценарій не є класичною програмною ін'єкцією, проте становить маніпуляцію бізнес-логікою. Збереження суперечливого стану може ускладнити подальшу перевірку повноважень і призвести до неоднозначного визначення користувачів, яким дозволено перегляд секретного значення. Тому нормалізація взаємозалежних параметрів повинна виконуватися на сервері незалежно від стану елементів інтерфейсу.

У класі *SecretForm* для цього перевизначено метод *clean()*. Після виконання стандартної валідації Django метод аналізує вибраний тип доступу та вилучає значення полів, які не повинні використовуватися в цьому режимі. Відповідний фрагмент наведено в лістингу 3.4.

Лістинг 3.4 – Серверна нормалізація параметрів доступу у формі *SecretForm*

```
def clean(self) -> dict[str, object]:
    cleaned_data: dict[str, object] = super().clean()
    access_type: str = str(
        cleaned_data.get("access_type") or "everyone"
    )

    if access_type == "everyone":
        cleaned_data["assigned_role"] = ""
        cleaned_data["assigned_user"] = None
    elif access_type == "role":
        cleaned_data["assigned_user"] = None
    elif access_type == "user":
        cleaned_data["assigned_role"] = ""

    return cleaned_data
```

Для перевірки сценарію VAL-01 було сформовано декілька POST-запитів із різними комбінаціями параметрів. Запити надсилалися від імені автентифікованого користувача з чинним CSRF-токеном, що дало змогу дослідити саме серверну бізнес-логіку форми, а не механізм захисту від CSRF.

Перший тестовий запит містив тип доступу *everyone*, але разом із ним передавалися значення *assigned_role=ADMIN* та ідентифікатор у полі *assigned_user*. Після оброблення форми обидва додаткові параметри були очищені. У базі даних збережено лише режим загального доступу без зв'язку з конкретною роллю або користувачем.

```
access_type=everyone
assigned_role=ADMIN
assigned_user=4
```

У другому сценарії для типу доступу *role* одночасно передавалися роль *ENGINEER* та ідентифікатор окремого користувача. Метод *clean()* зберіг значення ролі, але примусово встановив *assigned_user=None*.

```
access_type=role
assigned_role=ENGINEER
assigned_user=4
```

У третьому сценарії було вибрано тип *user*, але до запиту додатково включено роль *MANAGER*. Після серверної обробки роль була очищена, а зв'язок із вибраним користувачем збережено.

```
access_type=user
assigned_role=MANAGER
assigned_user=4
```

Перевірка показала, що реалізований метод успішно усуває суперечливі значення, передані в обхід користувацького інтерфейсу. Водночас було виявлено обмеження: поточна реалізація не перевіряє наявності обов'язкового пов'язаного значення. Сервер може прийняти режим *role* без вибраної ролі або режим *user* без вибраного користувача. У такому стані секрет формально створюється, але не має коректно визначеного адресата доступу.

Отже, сценарій VAL-01 пройдено частково. Нормалізація конфліктних полів виконується правильно, проте для забезпечення повної логічної цілісності форми необхідно доповнити метод *clean()* перевітками обов'язковості *assigned_role* та *assigned_user*. Відповідне вдосконалення і результати повторного тестування доцільно навести в підрозділі 3.4, присвяченому усуненню виявлених недоліків.

3.2.3 Перевірка захисту від SQL-ін'єкцій, XSS та CSRF

Окремий етап дослідження був спрямований на перевірку базових механізмів Django, які забезпечують захист від SQL-ін'єкцій, міжсайтового скриптингу та підроблення міжсайтових запитів. На відміну від об'єктного контролю доступу, ці механізми переважно реалізуються на рівні ORM, шаблонізатора та проміжного програмного забезпечення фреймворку.

Перевірка поєднувала статичний аналіз програмного коду з передаванням спеціально сформованих тестових значень. Під час статичного аналізу в розроблених модулях не виявлено використання методів *raw()*, *extra()*, класу *RawSQL*, ручної конкатенації SQL-запитів, функції *mark_safe()*, шаблонного фільтра *safe* або декоратора *csrf_exempt*. Це зменшує ймовірність обходу стандартних механізмів захисту Django, однак не скасовує необхідності практичної перевірки відповідних маршрутів [8].

Перевірка захисту від SQL-ін'єкцій

Робота з даними модулів *Tasks* і *Vault* виконується через Django ORM. Під час використання методів *filter()*, *get()* і *save()* значення передаються драйверу бази даних окремо від структури SQL-запиту. Завдяки цьому введений користувачем текст не повинен інтерпретуватися PostgreSQL як виконувана SQL-конструкція. Для перевірки сценарію SQL-01 у текстове поле опису завдання було передано тестове значення:

'OR 1=1 –

Після надсилання форми сервер створив один запис, а введена конструкція була збережена як звичайне текстове значення. Ознак зміни структури запиту,

повернення сторонніх записів або виникнення помилки PostgreSQL не зафіксовано. Отже, у перевіреному маршруті користувацьке значення не було інтерпретоване як частина SQL-команди.

Водночас отриманий результат стосується лише фактично дослідженої реалізації. Використання Django ORM не гарантує автоматичного захисту майбутніх фрагментів коду, якщо в них застосовуватимуться необроблені SQL-запити або ручне формування SQL-рядків [8].

Перевірка захисту від міжсайтового скриптингу

Міжсайтовий скриптинг становить загрозу для полів, значення яких вводяться одним користувачем, зберігаються у базі даних і надалі відображаються іншим користувачам. У розробленому сайті до таких полів належать назва й опис завдання, назва секрету та його опис.

Для перевірки сценарію XSS-01 у поле опису тестового завдання було введено маркер:

```
<script>alert('XSS_TEST')</script>
```

Після збереження запису сторінку завдання було відкрито повторно. Браузер не виконав JavaScript-код і не відобразив діалогове вікно. Введене значення було виведено як текст, оскільки шаблонізатор Django замінив спеціальні символи їх безпечними HTML-поданнями. У шаблоні значення опису виводиться стандартною конструкцією:

Лістинг 3.5 – Виведення опису завдання засобами шаблонізатора Django

```
<dd class="col-sm-9">
  {{ task.description }}
</dd>
```

У цьому фрагменті не використано фільтр *safe* та не вимкнено автоматичне екранування. Тому значення, отримане з бази даних, не включається до HTML-документа як виконуваний код. На рис. 3.3 наведено результат перевірки автоматичного екранування XSS-маркера.

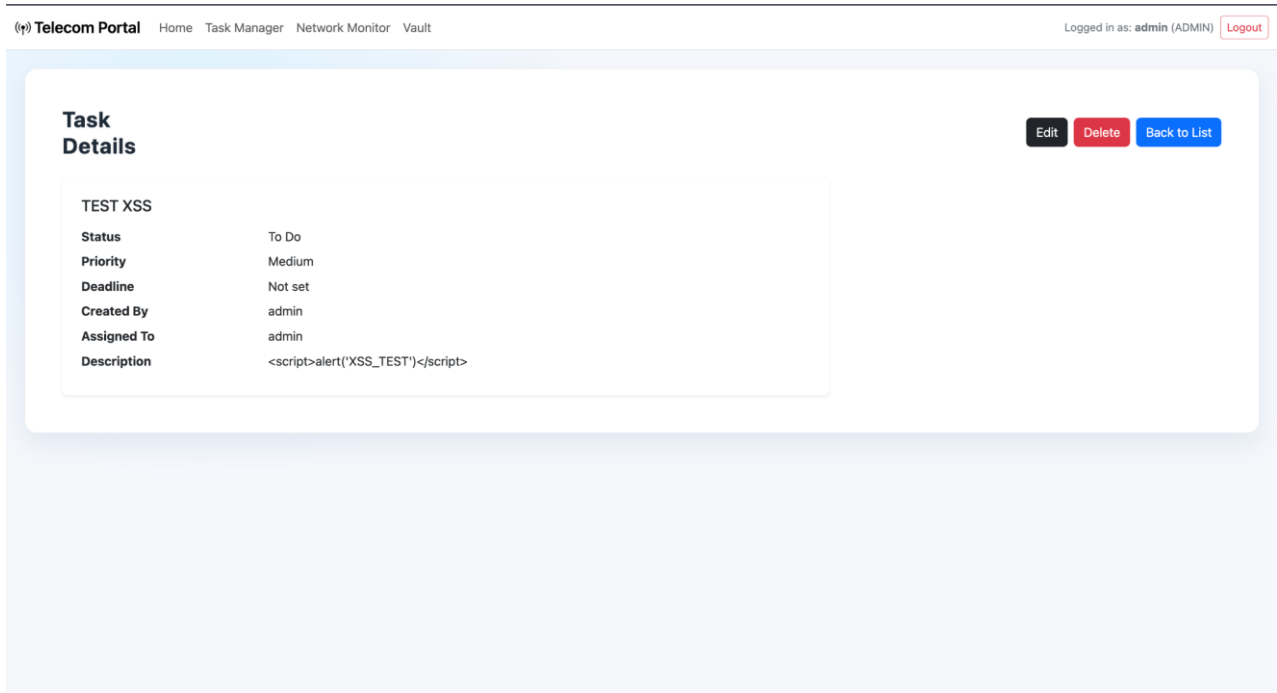


Рис. 3.3 – Результат перевірки автоматичного екранування XSS-маркера

Перевірка захисту від CSRF

Захист від підроблення міжсайтових запитів у проєкті реалізовано компонентом *CsrfViewMiddleware*, підключеним у конфігурації MIDDLEWARE. Усі HTML-форми, які змінюють стан системи, використовують метод POST і містять шаблонний тег `{% csrf_token %}`.

Наприклад, форма видалення завдання має такий вигляд:

Лістинг 3.6 – Використання CSRF-токена у формі видалення завдання

```
<form
  action="{% url 'tasks:task_delete' task.pk %}"
  method="post"
>
  {% csrf_token %}
  <button type="submit">Delete</button>
</form>
```

Для перевірки сценарію CSRF-01 було сформовано POST-запит на видалення тестового завдання без CSRF-cookie та без параметра `csrfmiddlewaretoken`. Сервер відхилив такий запит зі статусом HTTP 403, а відповідний запис залишився у базі даних.

Після цього той самий запит було виконано через штатну форму з чинним CSRF-токеном. За умови наявності необхідних повноважень операція завершилася успішно. Це підтверджує, що механізм не блокує легітимні дії користувача, але відхиляє запити, для яких неможливо підтвердити походження зі сторінки корпоративного сайту. Результати проведених перевірок узагальнено в таблиці 3.3.

Таблиця 3.3

Результати перевірки SQLi, XSS та CSRF

Код	Тестове значення або дія	Очікуваний результат	Фактичний результат
SQL-01	OR 1=1 -- у текстовому полі	Значення обробляється як текст	SQL-код не виконано, значення збережено як текст
XSS-01	<script>alert('XSS_TEST')</script>	Сценарій не виконується	Значення екрановано шаблонізатором
CSRF-01	POST-запит без CSRF-токена	HTTP 403, об'єкт збережено	Запит відхилено, об'єкт не видалено
CSRF-02	POST-запит із чинним токеном	Дозволена операція виконується	Штатна операція виконана

Проведені тести підтвердили коректну роботу параметризованих запитів Django ORM, автоматичного екранування шаблонних значень і CSRF middleware у досліджених маршрутах. Отриманий результат не доводить абсолютної відсутності відповідних класів вразливостей у будь-яких майбутніх компонентах, але підтверджує стійкість фактично перевірених форм і представлень.

3.2.4 Аналіз конфігурації розгортання Django

Захищеність веб додатку залежить не лише від програмної логіки, а й від параметрів середовища розгортання. Для перевірки конфігурації застосовано вбудовану команду Django:

`python manage.py check --deploy`

Команда аналізує налаштування, що мають значення для експлуатації системи у виробничому середовищі [8; 15]. За результатами перевірки виявлено шість попереджень, наведених у таблиці 3.4.

Таблиця 3.4

Результати первинного аналізу конфігурації Django

Параметр	Виявлений недолік	Потенційний ризик
DEBUG	режим зневадження активний	розкриття технічної інформації
SECRET_KEY	використовується неналежне резервне значення	зниження стійкості криптографічних механізмів Django
SECURE_SSL_REDIRECT	перенаправлення на HTTPS не ввімкнено	передавання даних незахищеним каналом
SECURE_HSTS_SECONDS	політика HSTS не налаштована	можливість примусового переходу на HTTP
SESSION_COOKIE_SECURE	прапорець Secure не встановлено	перехоплення сесійного cookie через HTTP
CSRF_COOKIE_SECURE	прапорець Secure не встановлено	передавання CSRF-cookie незахищеним каналом

Отримані результати відповідають ризику Security Misconfiguration. Виявлені параметри є допустимими лише для локального середовища розробки та не повинні переноситися до виробничої конфігурації.

Таким чином, сценарій CFG-01 на початковому етапі не пройдено: система потребує вимкнення режиму зневадження, використання надійного секретного ключа та налаштування захищеного передавання cookie через HTTPS. Відповідні зміни й результати повторної перевірки буде наведено в підрозділі 3.4.

3.3 Моделювання ланцюга експлуатації вразливостей

Під час аналізу початкової реалізації було виявлено поєднання двох недоліків: передавання введеної адреси командній оболонці та зберігання

ключа Fernet у вихідному коді. Їх послідовна експлуатація могла призвести від виконання системної команди до розкриття тестового секрету.

Перевірка проводилася в ізолюваному середовищі з використанням синтетичних даних. Наведений сценарій характеризує початкову вразливу версію модуля Network, а не його виправлений стан.

3.3.1 Експлуатація OS Command Injection

У початковій реалізації цільова адреса включалася в командний рядок без належної перевірки, після чого команда виконувалася з параметром *shell=True*.

Лістинг 3.7 – Початкова вразлива реалізація запуску утиліти ping

```
command = f"ping -c 3 {target}"
result = subprocess.run(
    command,
    shell=True,
    capture_output=True,
    text=True,
)
```

За такої реалізації командна оболонка інтерпретує метасимволи у значенні *target*. Для перевірки у поле мережевої адреси було передано:

8.8.8.8; ls -la

Символ `;` розділив вхідний рядок на дві команди. Після виконання `ping` сервер виконав `ls -la`, а перелік файлів повернувся у веб інтерфейс. Отже, сценарій INJ-01 у початковій реалізації завершився негативно: користувацьке значення вплинуло на структуру системної команди. Результат перевірки наведено на рис. 3.4.

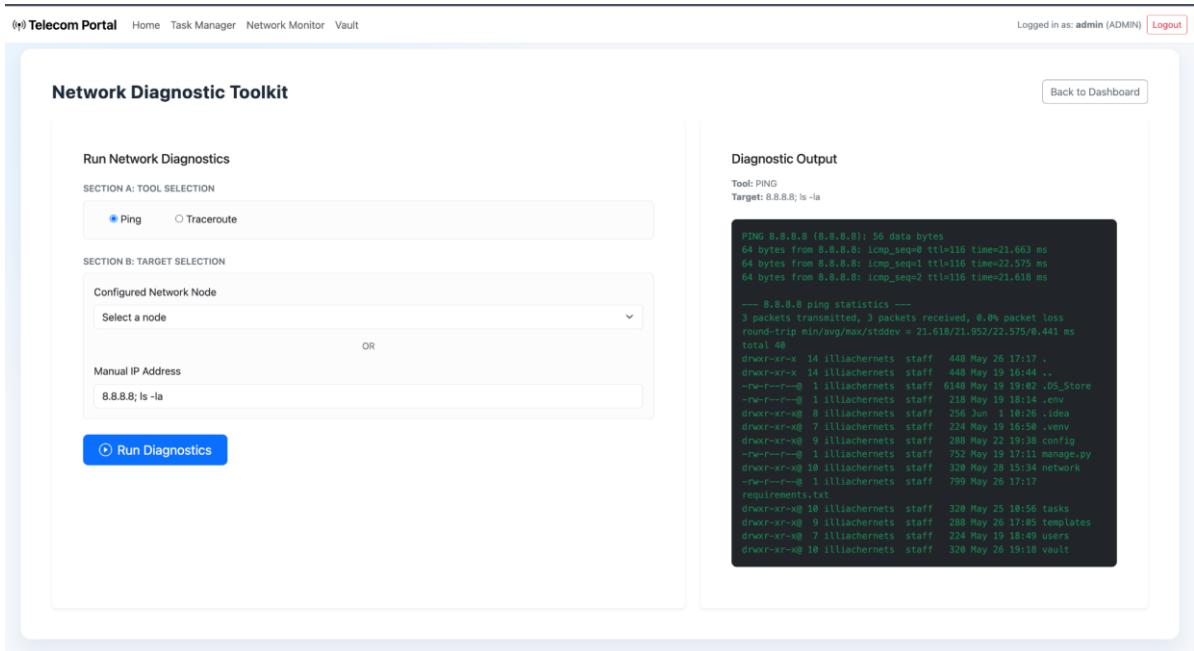


Рис. 3.4 – Результат експлуатації OS Command Injection у модулі Network

Виконані команди отримали лише права системного користувача, від імені якого працював процес Django. Проте цих прав було достатньо для читання доступних файлів проєкту.

3.3.2 Розкриття вихідного коду та криптографічного ключа

Після визначення структури каталогів тестовий сценарій було продовжено читанням файла криптографічної моделі:

8.8.8.8; cat vault/models.py

У відповіді сервера було виявлено константу FERNET_KEY, розміщену безпосередньо у вихідному коді. Такий недолік відповідає CWE-321, оскільки компрометація файла одночасно розкриває логіку дешифрування та ключовий матеріал. Результат перевірки наведено на рис 3.5.

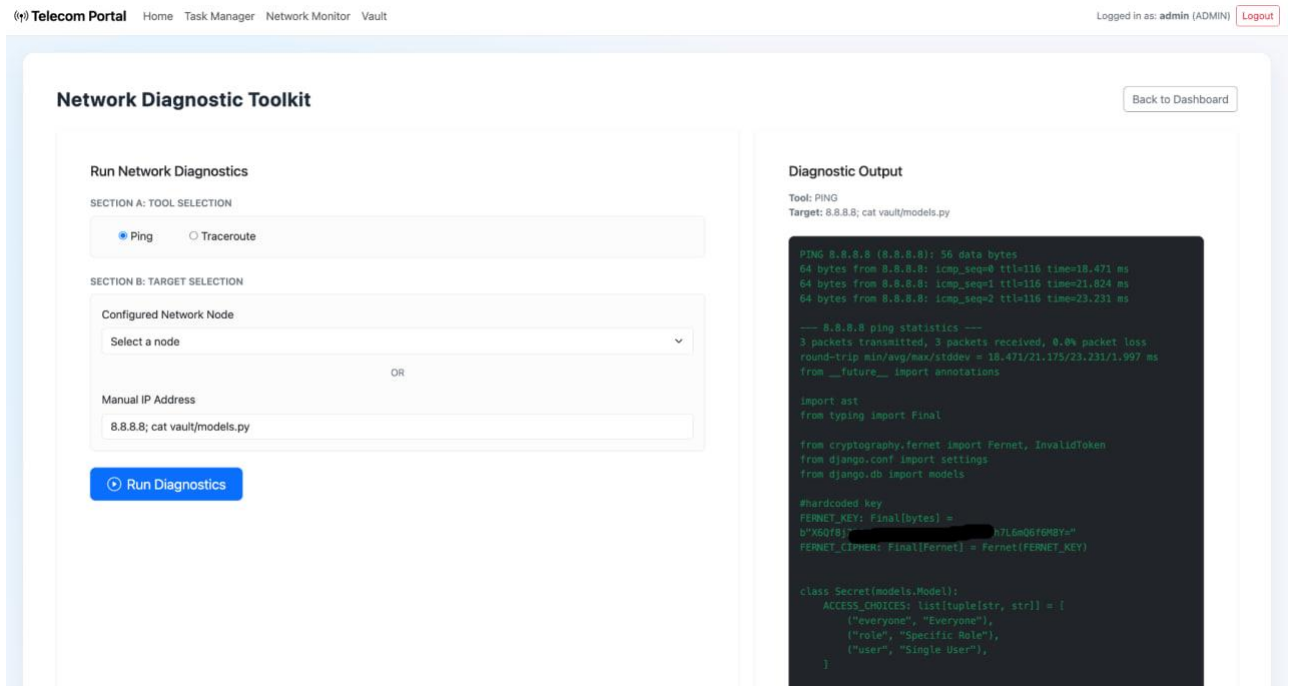


Рис. 3.5 – Розкриття жорстко закодованого ключа Fernet через Command Injection

Виявлений результат не свідчить про криптографічну слабкість Fernet. Причиною компрометації є неналежне керування ключем: шифротексти та засіб їх дешифрування стали доступними в межах одного ланцюга атаки [5; 14].

3.3.3 Офлайн-дешифрування тестового секрету

Для підтвердження наслідків атаки з лабораторної бази даних було отримано зашифрований токен спеціально створеного тестового секрету. Викрадений ключ і токен передано локальному скрипту.

Лістинг 3.8 – Офлайн-дешифрування тестового токена

```
from cryptography.fernet import Fernet
stolen_key = b"<тестовий ключ Fernet>"
stolen_token = b"<тестовий токен із лабораторної БД>"
cipher = Fernet(stolen_key)
decrypted_value = cipher.decrypt(stolen_token).decode("utf-8")
print(decrypted_value)
```

У результаті було відновлено початкове тестове значення, що показане на рис. 3.6. Таким чином, сценарій CRY-01 підтвердив наявність ключа у

вихідному коді, а сценарій CRY-02 показав можливість дешифрування токена після отримання відповідного ключа.

```
>>> from cryptography.fernet import Fernet
>>> stolen_key = b"X6Qf8j2B0wM4Q0E1Qq:~n5R1Qh7L6mQ6f6M8Y="
>>> stolen_token = b"gAAAAA~.Sxe09wbW3p-7dGHKPRUR6TH4ZDsbVvwPYmuES30lUp-zSwxERyr7VnkkwB4iEWfExJLTNS6nHzmTw=="
>>> cipher = Fernet(stolen_key)
>>> print(cipher.decrypt(stolen_token).decode("utf-8"))
test_secret
>>>
```

Рис. 3.6 – Результат офлайн-дешифрування тестового секрету

Проведений експеримент підтвердив, що наслідок Command Injection посилюється через жорстке кодування ключа. Тому захист має передбачати одночасно безпечний запуск системних процесів та відокремлення криптографічного ключа від вихідного коду.

3.4 Усунення виявлених вразливостей і повторна перевірка

За результатами практичного дослідження було сформовано захисні зміни для модулів Network і Vault, серверної валідації форм та конфігурації Django. Після рефакторизації повторно розглянуто початкові сценарії, щоб перевірити блокування атак і збереження штатної функціональності системи.

3.4.1 Усунення OS Command Injection

Причиною вразливості було формування команди як одного рядка та її виконання через системну оболонку. У виправленій реалізації аргументи передаються до `subprocess.run()` списком, а параметр `shell=False` виключає інтерпретацію метасимволів. Результат перевірки наведено на рис. 3.7.

Лістинг 3.9 – Безпечний запуск мережевих утиліт

```
def validate_target(target: str) -> bool:
    pattern = re.compile(r"^[a-zA-Z0-9.-]+$")
    return bool(pattern.fullmatch(target))

if not validate_target(target):
    output = "Security Alert: Invalid target format."
else:
```

```

if selected_tool == "traceroute":
    command = ["traceroute", "-n", target]
else:
    command = ["ping", "-c", "3", target]
result = subprocess.run(
    command,
    shell=False,
    capture_output=True,
    text=True,
    timeout=25,
    check=False,
)

```

Після внесення змін значення:

8.8.8.8; ls -la

було відхилено до запуску системного процесу.

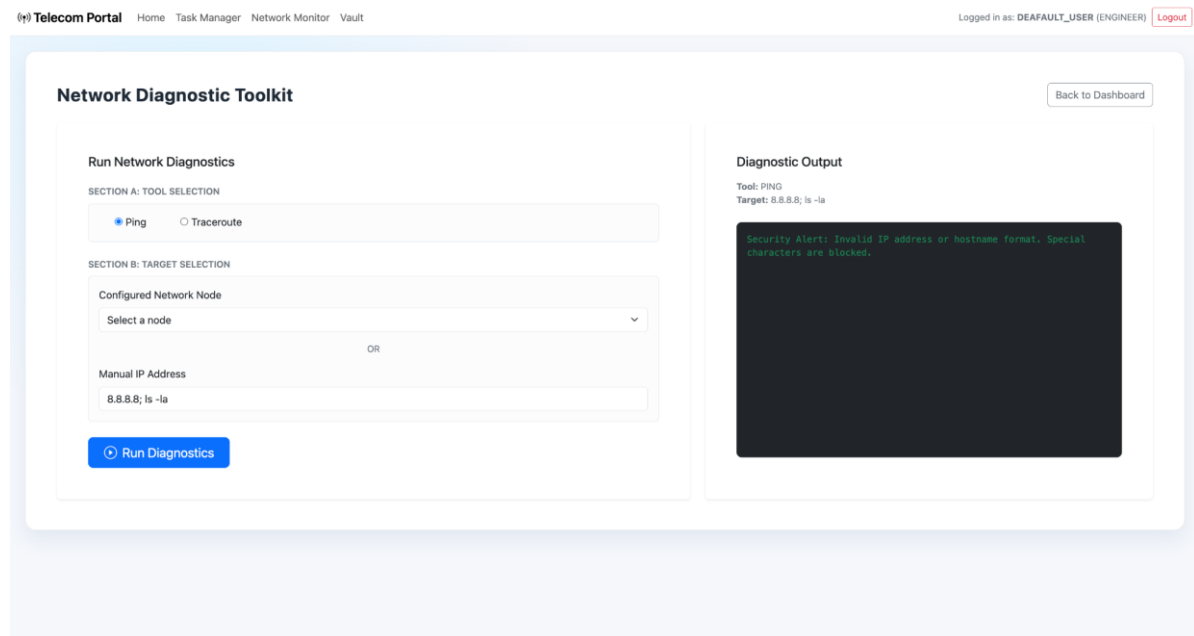


Рис. 3.7 – Блокування повторної спроби OS Command Injection

3.4.2 Усунення жорстко закодованого ключа

Для зменшення ризику розкриття ключа під час витоку вихідного коду константу `FERNET_KEY` вилучено з моделі. Значення завантажується зі змінної середовища, а запуск застосунку без ключа завершується контрольованою помилкою.

Лістинг 3.10 – Завантаження ключа Fernet зі змінної середовища

```
import os

from cryptography.fernet import Fernet
from django.core.exceptions import ImproperlyConfigured

fernet_key = os.getenv("VAULT_FERNET_KEY")

if not fernet_key:
    raise ImproperlyConfigured(
        "VAULT_FERNET_KEY is not configured."
    )
FERNET_CIPHER = Fernet(fernet_key.encode("utf-8"))
```

Після зміни читання `vault/models.py` більше не розкриває ключовий матеріал. Водночас цей захід не захищає від повного виконання команд у контексті процесу Django, оскільки застосунок повинен мати доступ до ключа. Тому ізоляція ключа застосовується разом з усуненням Command Injection [5; 14].

3.4.3 Посилення серверної валідації та конфігурації

Метод `SecretForm.clean()` було доповнено перевіркою обов'язкових значень для режимів `role` і `user`.

Лістинг 3.11 – Перевірка повноти параметрів доступу

```
if access_type == "role":
    cleaned_data["assigned_user"] = None

    if not cleaned_data.get("assigned_role"):
        self.add_error(
            "assigned_role",
            "Select a role for this access type.",
        )

elif access_type == "user":
    cleaned_data["assigned_role"] = ""

    if not cleaned_data.get("assigned_user"):
        self.add_error(
            "assigned_user",
            "Select a user for this access type.",
        )
```

Для виробничої конфігурації також визначено такі параметри:

DEBUG = False

SECURE_SSL_REDIRECT = True

SESSION_COOKIE_SECURE = True

CSRF_COOKIE_SECURE = True

SECURE_HSTS_SECONDS = 31536000

Значення *SECRET_KEY* повинно завантажуватися зі змінної середовища без небезпечного резервного значення. Параметри HTTPS та HSTS вмикаються лише після налаштування TLS на вебсервері або зворотному проксі.

3.4.4 Результати повторного тестування

Після рефакторизації було повторено сценарії, за якими на початковому етапі виявлено недоліки.

Таблиця 3.5

Порівняння результатів до та після рефакторизації

Сценарій	Стан до архітектурної рефакторизації	Результат повторного тестування
INJ-01	Успішне виконання ін'єктованої команди ОС (ls) в обхід веб-ізоляції через параметр shell=True.	Вектор атаки нейтралізовано: невалідні вхідні дані блокуються на етапі санітизації
CRY-01	Криптографічний ключ збережено у відкритому вигляді безпосередньо у вихідному коді моделі.	Ключ ізольовано у незалежних змінних середовища; ризик ексфільтрації усунено.
CRY-02	Успішне дешифрування викраденого токена бази даних за допомогою скомпрометованого ключа.	Цілісність криптографічного контуру відновлено: без доступу до оперативної пам'яті сервера дешифрування математично неможливе.
VAL-01	Можливість виникнення логічних колізій у БД шляхом передачі суперечливих параметрів доступу в обхід клієнтського JavaScript.y	Імплементовано примусову серверну санітизацію; суперечливі HTTP-запити автоматично відхиляються.
CFG-01	Виявлено попередження конфігурації ядра фреймворку.	Імплементовано сувору Production-конфігурацію
Штатний ping	Штатне функціонування утиліт ICMP Ping та Traceroute.	Функціональну працездатність повністю збережено за умови безпечного виклику підпроцесів (shell=False).

Повторна перевірка підтвердила блокування дослідженого сценарію Command Injection та усунення ключа з вихідного коду. Водночас залишковий ризик пов'язаний із доступом процесу Django до криптографічного матеріалу, можливістю зловживання дозволеними мережевими операціями та потребою належного налаштування HTTPS у виробничому середовищі.

Висновки

У даному розділі було проведено практичне дослідження інформаційної безпеки розробленого корпоративного сайту. Дослідження підтвердило, що попри ефективну протидію класичним веб загрозам за допомогою вбудованих механізмів фреймворку та об'єктного контролю доступу, стійкість системи критично залежить від низькорівневих архітектурних рішень. Шляхом моделювання ланцюга кібератаки було успішно експлуатовано вразливість впровадження команд операційної системи у підсистемі мережевої діагностики. Експлуатація дозволила обійти механізми веб-ізоляції та здійснити ексфільтрацію вихідного коду, що призвело до розкриття жорстко закодованого ключа та повного компрометування криптографічного модуля *Vault*.

Ключовим етапом розділу стала успішно проведена архітектурна рефакторизація вразливого вихідного коду. Впровадження суворої санітизації вхідних параметрів за допомогою регулярних виразів, перехід на безпечний виклик підпроцесів без використання системної оболонки та ізоляція криптографічних констант у незалежних змінних середовища повністю нейтралізували виявлені вектори атак. Виконані заходи призвели розроблений сайт у повну відповідність до актуальних вимог безпечного життєвого циклу програмного забезпечення, довівши його готовність до безпечної експлуатації в реальному телекомунікаційному середовищі.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання розробки та комплексного практичного дослідження інформаційної безпеки корпоративного телекомунікаційного сайту, побудованого на базі високорівневого фреймворку Django. За результатами проведеного інженерного проєктування та практичного моделювання кібератак можна зробити наступні висновки:

1. Проєктування архітектури та баз даних. Доведено доцільність застосування архітектурного шаблону MVT у поєднанні з системою керування базами даних PostgreSQL для телекомунікаційних потреб. Впровадження механізму багатоверсійного керування конкурентним доступом забезпечило стабільну паралельну обробку запитів від співробітників NOC без ризику транзакційних блокувань.
2. Розробка функціональних підсистем. Успішно програмно реалізовано підсистему управління завданнями та інтегровано гнучку рольову модель контролю доступу. Реалізація гранулярної перевірки прав на рівні об'єктів забезпечила захист від загроз горизонтальної та вертикальної ескалації привілеїв, гарантуючи ізоляцію конфіденційних інфраструктурних даних між Адміністраторами, Менеджерами та Інженерами.
3. Захист криптографічної інформації. Спроектовано підсистему Vault для безпечного управління інфраструктурними секретами. Застосування стандарту симетричного шифрування AES-128 дозволило реалізувати концепцію захисту даних у стані спокою. Водночас практично доведено, що стійкість алгоритму критично залежить від ізоляції криптографічного ключа: жорстке кодування ключа у вихідному коді зводить нанівець математичну складність шифрування.
4. Аналіз безпеки та експлуатація вразливостей. На базі розробленого модуля мережевої діагностики змодельовано ланцюг кібератаки типу OS Command Injection. Практично продемонстровано, як неконтрольована

взаємодія з оболонкою операційної системи дозволяє зломиснику обійти веб-ізоляцію, здійснити розвідку файлової системи сервера, ексфільтрувати вихідний код модулів безпеки та викрасти криптографічний ключ.

5. Архітектурна рефакторизація. За результатами практичного дослідження проведено патч-менеджмент та усунено виявлені архітектурні дефекти. Впровадження суворої валідації вхідних даних, відмова від системної оболонки на користь параметризованих викликів процесу та винесення криптографічних констант у змінні середовища операційної системи дозволило повністю ліквідувати вектори ін'єкцій.

Результати роботи мають високий ступінь практичної значущості і можуть бути імplementовані в інфраструктуру інтернет-провайдерів середнього та великого масштабу для оптимізації процесів моніторингу та безпечного управління життєвим циклом інфраструктурних секретів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What are Application Security Risks? (OWASP Top 10:2025) [Електронний ресурс] / OWASP Foundation. – 2025. – Режим доступу до ресурсу: https://owasp.org/Top10/2025/0x02_2025-What_are_Application_Security_Risks/ (дата звернення: 06.05.2026).
2. OWASP Top 10:2025 – A01 Broken Access Control [Електронний ресурс] / OWASP Foundation. – 2025. – Режим доступу до ресурсу: https://owasp.org/Top10/2025/A01_2025-Broken_Access_Control/ (дата звернення: 06.05.2026).
3. OWASP Application Security Verification Standard 5.0.0 [Електронний ресурс] / OWASP Foundation. – 2025. – Режим доступу до ресурсу: <https://owasp.org/www-project-application-security-verification-standard/> (дата звернення: 07.05.2026).
4. OWASP Top 10:2025 – A04 Cryptographic Failures [Електронний ресурс] / OWASP Foundation. – 2025. – Режим доступу до ресурсу: https://owasp.org/Top10/2025/A04_2025-Cryptographic_Failures/ (дата звернення: 26.05.2026).
5. Barker E. Recommendation for Key Management: Part 1 – General, Revision 5 [Електронний ресурс] / E. Barker // NIST Special Publication 800-57 Part 1. – 2020. – Режим доступу до ресурсу: <https://doi.org/10.6028/NIST.SP.800-57pt1r5> (дата звернення: 12.05.2026).
6. Hoffman A. Web Application Security: Exploitation and Countermeasures for Modern Web Applications / A. Hoffman. – O'Reilly Media, 2020. – 327 p.
7. Stuttard D., Pinto M. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws / D. Stuttard, M. Pinto. – 2nd ed. – John Wiley & Sons, 2011. – 912 p.
8. Security in Django 5.2 [Електронний ресурс] / Django Software Foundation. – Режим доступу до ресурсу: <https://docs.djangoproject.com/en/5.2/topics/security/> (дата звернення: 13.05.2026).

9. Zalewski M. The Tangled Web: A Guide to Securing Modern Web Applications / M. Zalewski. – No Starch Press, 2011. – 320 p.
10. Content Security Policy (CSP) [Электронный ресурс] / MDN Web Docs. – Режим доступа до ресурсу: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP> (дата звернення: 25.05.2026).
11. Spilca L. Spring Security in Action / L. Spilca. – Manning Publications, 2020. – 560 p.
12. Node.js Security Best Practices [Электронный ресурс] / OpenJS Foundation. – Режим доступа до ресурсу: <https://nodejs.org/en/learn/getting-started/security-best-practices> (дата звернення: 26.05.2026).
13. Production Best Practices: Security [Электронный ресурс] / Express.js. – Режим доступа до ресурсу: <https://expressjs.com/en/advanced/best-practice-security.html> (дата звернення: 26.05.2026).
14. Fernet (Symmetric Encryption) [Электронный ресурс] / Cryptography Project. – Режим доступа до ресурсу: <https://cryptography.io/en/latest/fernet/> (дата звернення: 27.05.2026).
15. Django 5.2 documentation [Электронный ресурс] / Django Software Foundation. – Режим доступа до ресурсу: <https://docs.djangoproject.com/en/5.2/> (дата звернення: 29.05.2026).
16. Vincent W. S. Django for Beginners: Build Modern Web Applications with Python / W. S. Vincent. – 5th ed. – Still River Press, 2024. – 357 p.
17. Feldroy D., Feldroy A. Two Scoops of Django 3.x: Best Practices for the Django Web Framework [Электронный ресурс] / D. Feldroy, A. Feldroy. – 5th ed. – Two Scoops Press, 2021. – Режим доступа до ресурсу: <https://www.feldroy.com/two-scoops-of-django> (дата звернення: 01.06.2026).
18. Anderson R. Security Engineering: A Guide to Building Dependable Distributed Systems / R. Anderson. – 3rd ed. – John Wiley & Sons, 2020. – 1232 p.
19. File Locking and Concurrency in SQLite Version 3 [Электронный ресурс] / SQLite. – Режим доступа до ресурсу: <https://www.sqlite.org/lockingv3.html> (дата звернення: 01.06.2026).

20. Ciolli G., Mejías B., Angelakos J., Kumar V., Riggs S. PostgreSQL 16 Administration Cookbook: Solve Real-World Database Administration Challenges with 180+ Practical Recipes and Best Practices / G. Ciolli, B. Mejías, J. Angelakos, V. Kumar, S. Riggs. – 1st ed. – Packt Publishing, 2023. – 636 p.
21. Elmasri R., Navathe S. B. Fundamentals of Database Systems / R. Elmasri, S. B. Navathe. – 7th ed. – Pearson, 2016. – 1242 p.
22. PostgreSQL 16 Documentation: Data Types [Электронный ресурс] / PostgreSQL Global Development Group. – Режим доступа до ресурсу: <https://www.postgresql.org/docs/16/datatype.html> (дата звернення: 02.06.2026).
23. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach, Global Edition / J. F. Kurose, K. W. Ross. – 8th ed. – Pearson, 2021. – 800 p.
24. Fall K. R., Stevens W. R. TCP/IP Illustrated, Volume 1: The Protocols / K. R. Fall, W. R. Stevens. – 2nd ed. – Addison-Wesley Professional, 2011. – 1056 p.
25. Stallings W. Cryptography and Network Security: Principles and Practice / W. Stallings. – 8th ed. – Pearson, 2020. – 768 p.
26. OWASP Web Security Testing Guide, Version 4.2 [Электронный ресурс] / OWASP Foundation. – 2020. – Режим доступа до ресурсу: <https://owasp.org/www-project-web-security-testing-guide/v42/> (дата звернення: 03.06.2026).