Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «магістр»

на тему:

КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ ІНФОРМАЦІЙНИХ ПРОЦЕСІВ ТА ПОВЕДІНКИ СИСТЕМИ КІБЕРЗАХИСТУ

Виконав:
здобувач 2 курсу
групи М-КН-21
спеціальності 122 «Комп'ютерні науки»
Волощук Владислав Олександрович

Науковий керівник:
кандидат технічних наук,
доцент Сінчук Алеся Михайлівна

Рівне – 2024

ЗМІСТ

ВСТУП

Актуальність роботи. З розвитком цифрових технологій і поширенням кіберзагроз, питання кібербезпеки стає все більш актуальним. Статистичні дані свідчать про щорічне зростання кількості атак, зокрема з використанням вірусів, DDoS-атак, витоків даних та інших вразливостей [1, 2]. Особливу увагу викликає проблема захисту конфіденційної інформації, яка зберігається в цифрових системах. Як зазначають автори наукових досліджень, таких як «Класифікація кіберзагроз» та «Методи та засоби захисту інформації» [15, 18], сучасні атаки стають дедалі складнішими, що вимагає інтеграції інтелектуальних технологій для протидії загрозам.

Сьогодні особливу важливість має створення систем, здатних оперативно діагностувати потенційні загрози або аналізувати інші дані, зокрема у сфері охорони здоров’я. Симптоматична діагностика на основі великих баз даних і автоматизованих алгоритмів є одним із перспективних напрямків, оскільки вона дозволяє швидко отримувати попередні діагнози, забезпечуючи економію часу та ресурсів. Водночас питання забезпечення кібербезпеки таких систем залишається відкритим і потребує подальших досліджень.

Метою роботи є розробка та впровадження веб-додатка для симптоматичної діагностики, який використовує інтелектуальні алгоритми для аналізу введених симптомів і забезпечує високий рівень кібербезпеки даних користувачів. Дослідження спрямоване на вдосконалення алгоритмів діагностики та розробку ефективних механізмів захисту даних, включаючи шифрування, захист від атак CSRF, XSS і SQL-ін'єкцій.

Для досягнення сформульованої мети були поставлені такі **завдання**:

- 1. Провести аналіз наукових джерел з історії розвитку кібербезпеки та класифікації загроз.
- 2. Дослідити еволюцію методів захисту інформації, зокрема в контексті веб-додатків.

- 3. Розробити алгоритм діагностики, який враховує вагу симптомів для підвищення точності.
- 4. Реалізувати клієнтську та серверну частини веб-додатка з використанням сучасних технологій (Python, Flask, PostgreSQL).
- 5. Забезпечити інтеграцію механізмів кібербезпеки, таких як Google OAuth 2.0, шифрування даних, обмеження запитів.

Об’єкт дослідження: процеси та технології інтеграції діагностичних алгоритмів у веб-додатках із забезпеченням їхньої кібербезпеки.

Предмет дослідження: алгоритм діагностики за симптомами та засоби забезпечення захисту інформації у веб-додатку.

Методи дослідження:

- теоретичні методи: аналіз літературних джерел з кібербезпеки та методів діагностики, моделювання алгоритмів;
- емпіричні методи: проєктування, розробка та тестування веб-додатка;
- практичні методи: впровадження механізмів захисту даних, проведення аналізу безпеки системи.

Практичне значення дослідження. Результати дослідження можуть бути використані для розробки подібних систем, що інтегрують алгоритми аналізу даних із засобами кіберзахисту. Створений веб-додаток може стати основою для впровадження в медичні заклади, надаючи базові рекомендації щодо стану здоров’я пацієнтів.

Апробація та впровадження результатів дослідження. Основні теоретичні та практичні результати дослідження були представлені на XVII Всеукраїнській науково-практичній конференції «Інформаційні технології в професійній діяльності» (Рівне, 5 листопада 2024 р.).

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел та додатків. У першому розділі проведено аналіз сучасних літературних джерел з кібербезпеки, розглянуто виникнення та

еволюцію кіберзагроз, а також основні етапи розвитку методів захисту інформації. Другий розділ містить огляд існуючих підходів до захисту інформаційних систем, розглядає сучасні виклики у сфері кібербезпеки, а також аналізує нормативно-правову базу, яка регулює захист інформації. Третій розділ присвячено розробці та реалізації інтелектуальної системи діагностики, включаючи архітектуру проєкту, опис алгоритмів, вибір технологій і впровадження кіберзахисних механізмів. Загальний обсяг роботи становить 59 сторінок. Вона містить 33 рисунків. Список використаних джерел включає 32 найменувань.

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМИ ДОСЛІДЖЕННЯ ТА ІСТОРІЯ РОЗВИТКУ КІБЕРБЕЗПЕКИ

1.1. Аналіз літературних джерел

У процесі виконання роботи було використано широкий спектр літературних джерел, які можна класифікувати за кількома групами: історичні матеріали, технічні статті, документація та наукові дослідження. Кожна з цих категорій зробила свій внесок у розробку, обґрунтування та впровадження методів і технологій, використаних у проєкті.

Дослідження з історії кібербезпеки та комп'ютерних загроз стало базою для визначення основних тенденцій і закономірностей у розвитку кіберзагроз. Наприклад, стаття «The History of the First Computer Virus on Windows, Mac, and Linux» [1] надала корисну інформацію про перші комп’ютерні віруси, їхню еволюцію та вплив на безпеку сучасних систем. Аналогічно, матеріали з «Події з історії кібербезпеки за 50 років» [2] висвітлили основні етапи розвитку кіберзахисту, зокрема ключові події, що змінили підходи до забезпечення безпеки.

Джерела з історії криптографії, такі як «A Short History of Cryptography» [4] та «RSA Algorithm in Cryptography» [5], дали можливість детальніше розглянути еволюцію методів шифрування, а також розробку сучасних алгоритмів, таких як RSA та AES. Це стало основою для впровадження алгоритмів шифрування даних користувачів та історії чату в нашій системі.

Матеріали на тему класифікації кіберзагроз і їхніх методів протидії, як-от «What is hacktivism?» [14] і »Класифікація кіберзагроз та їх легітимація у нормативно-правових актах України» [15], дозволили детально вивчити характер атак і способи захисту, включаючи методи виявлення, попередження та нейтралізації загроз. Ці знання лягли в основу підпунктів, присвячених захисту від XSS, CSRF та інших атак.

Розробка програмного забезпечення базувалася на сучасній технічній документації. Наприклад:

- Документація Flask [24] та Flask-WTF [29] забезпечила інтеграцію веб-фреймворку та механізмів захисту від CSRF.

- PostgreSQL Documentation [26] надала інструкції щодо налаштування бази даних і захисту від SQL-ін'єкцій.

- Flask-Limiter Documentation [30] допомогла впровадити механізм обмеження запитів для запобігання DDoS-атакам.

Статті, такі як «What is EDR vs. XDR?» [21], надали сучасний огляд технологій виявлення та протидії кіберзагрозам, таких як EDR (Endpoint Detection and Response) та XDR (Extended Detection and Response). Ці знання вказали на можливі напрямки для подальшого розвитку системи.

Інструкції з розробки динамічних вебінтерфейсів, представлені на MDN Web Docs [27] та jQuery API Documentation [28], спростили реалізацію клієнтської частини додатка. Застосування сучасних стандартів HTML5, CSS3 та JavaScript забезпечило адаптивність інтерфейсу та зручність використання.

Матеріали, такі як «Industrial Network Security» [17] та «Методи та засоби захисту інформації» [18], дозволили оцінити сучасні тенденції в галузі кібербезпеки.

Усі використані джерела мали високий рівень актуальності та наукової цінності. Онлайн-матеріали (документація, статті) надали практичні рекомендації, тоді як наукові статті та книги забезпечили теоретичну базу. Однак було б корисно розширити аналіз за рахунок більшої кількості досліджень з української нормативно-правової бази, що регулює кіберзахист, для врахування локальних специфік.

1.2. Виникнення та еволюція кіберзагроз

Кіберзагрози виникли з самого початку розвитку комп'ютерних технологій та мереж. Перші кіберзагрози були досить примітивними – комп'ютерні віруси та

черв'яки, які створювалися переважно для демонстрації можливостей або жарти.

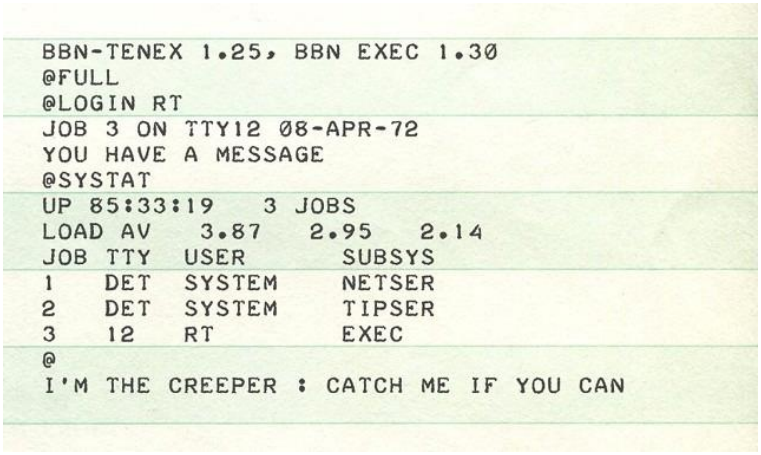

Рисунок 1.1. Вірус Creeper

Одним із перших відомих вірусів був Creeper, створений у 1971 році, який не завдавав ніякої шкоди, а лише друкував повідомлення (рис 1.1).

Серед деяких із найперших комп’ютерних вірусів цей вважається першим, хто вийшов за межі комп’ютерної системи, на якій він був розроблений. «Elk Cloner» був створений Річардом Скрентою, 15-річним старшокласником, який зробив вірус для Apple II системи десь близько 1982 року. Вірус зміг поширитися, заразивши операційну систему на Apple II, яка зберігалася на дискетах. Як і в більшості перших комп’ютерних вірусів пошкодження, спричинені Elk Cloner, були невисокими, але багатьох користувачів дратувало відображення вірша (рис. 1.2), який з’являвся під час кожного 50-го завантаження. [1]

Рисунок 1.2. Вірус Elk Cloner

Вірус Морріса – 1988 рік. До появи цього комп'ютерного вірусу інші віруси поширювалися в набагато менших масштабах, а цеq справді поширився по всьому

світу. Його розробив аспірант Корнельського університету, який також був сином високопоставленого державного спеціаліста з комп’ютерної безпеки. Цей комп’ютерний вірус зумів заразити близько 6000 університетських і військових машин, включно з комп’ютерами Інституту досліджень NASA де це повністю паралізувало роботу машин. Ця атака привела до масштабного зараження всієї мережі ARPANET (перша мережа передачі пакетів, яка стала попередницею сучасного інтернету), і саме після цього інциденту розробники придумали зашифровані паролі, а також паузи в разі неправильно введеного пароля.

Разом з цим більшість країн почали приймати закони про неправомірне використання комп’ютерів, зловживання і шахрайство з використанням комп’ютерів.

1999 рік – Вірус Melissa інфікував користувачів через Microsoft Outlook, що призвело до збитків у розмірі близько 1,2 мільярда доларів.

2000 рік – Вірус ILOVEYOU заразив понад 500 000 систем, переписував файли користувачів, завдаючи збитків на суму 15 мільярдів доларів.

У 2002 році була влаштована потужна DDoS-атака на 13 кореневих DNS-серверів, п'ять з яких були виведені з ладу. Це була перша спроба вимкнути Інтернет.

2007 рік. Під час політичної кризи в Естонії, бути здійснено масовану DDoS-атаку на державні сайти, банки та медіа. Це паралізувало країну на кілька тижнів , показало вразливість національної інфраструктури до кібератак і стало основним дзвінком до початку обговорення кібербезпеки в НАТО.

У 2010 був запущений хробак Stuxnet, що був першою відомою кібератакою, стпрямованою на фізичні об’єкти. Він заразив іранські ядерні об’єкти, викликавши пошкодження багатьох елементів станцій, а також центрифуг для збагачення урану. Вважається, що його розробили спільно США та Ізраїль.

Також не варто забувати про витіки даних, оскільки вони відбуваються ледь не щотижня і цьому ніяк не запобігти. Одним з найбільших можна вважати витік даних Yahoo у 2013 році, де хакери викрали понад 3 мільярдів облікових записів. Хоча витіки відбувалися і раніше, саме ця подія стала початком хвилі масових крадіжок

даних великих компаній. [2]

Одним з найвідоміших вірусів усіх часів став WannaCry, який у 2017 році використав вразливість у Windows для глобального поширення, вивівши з ладу понад 200000 комп’ютерів у 150 країнах. Ця атака сильно зачепила лікарні, банки та урядові установи, завдавши збітків на понад 4 мільярди доларів.

А з початком повномасштабного вторгнення росії в Україну багато українських урядових систем та пристроїв звичайних людей зазнали численних атак, зокрема вірусів-вимагачів і DDoS. Також атак зазнала і критична інфраструктура, зокрема енергетична.

А на сьогодні кіберзагрози еволюціонували до багатошарових атак зі штучним інтелектом. Багато де використовується соціальна інженерія, через необачність простого люду. Зловмисники активно атакують хмарні сервіси, криптовалютні платформи та багато пристроїв Інтернет речей.

1.3. Основні етапи розвитку методів захисту інформації

Початкові методи захисту інформації

Захист інформації має дуже довгу історію, що починається задовго до першого комп’ютера чи навіть математичного концепту. Методи захисту пройшли шлях від простих писемних чи механічних до сучасних високотехнологічних систем.

Історично захист інформації почався ще з античних часів, і найвідоміший шифр тих часів – це шифр Цезаря, що був простив алгоритмом зсуву літер в алфавіті. Використовувався приблизно з 50 року до н.е. Наприклад, зсув на три позиціє відбувався приблизно отак:

АБВГҐДЕЄЖЗИІЇЙКЛМНОПРСТУФХЦЧШЩЬЮЯ

ЬЮЯАБВГҐДЕЄЖЗИІЇЙКЛАМОПРСТУФХЦЧШЩ

Крім того використовувалися безліч схожих простих шифрів, серед яких шифр Atbash. У цьому методі остання літера алфавіту замінюється першою, і навпаки. Або ж шифр Skytale також відомий як «штатний шифр». Текст, який хотіли

захистити, записували на тонкому аркуші папірусу, обгорнутого навколо посоха. Папір потім розгортають, і щоб прочитати зашифрований текст треба було знову обгорнути папіпус на палицю однакового розміру (рис. 1.3).

Рисунок 1.3. Зразок шифру Skytale

Існують також поліалфавітні шифри. Найвідоміший – Віженерів шифр. За основу там взято той же самий шифр Цезаря, але використовується він в багатоалфавітній підстановці, що робить його складнішим для злому, ніж простий шифр з фіксованим зсувом. Також головною відмінністю є наявність ключа – слово або фраза. Використовувався з XVI століття.

Також мав місце фізичний захист, що використовується і донині. Інформація зберігалася у вигляді паперових документів, доступ до яких обмежувався через замки, сейфи, охорону тощо.

Механічні та електромеханічні пристрої

У XIX і до середина XX століття почали набирати популярність механічні та електромеханічні шифрувальні пристрої. Найвідоміший такий пристрій – машина Enigma (1920-ті роки), що використовувалася німецькими військовими і на той час створювала практично незламний шифр. Як підтвердження цьому, дешифровка повідомлень, зашифрованих на Enigma стала можливою лише під час другої світової війни, коли група вчених, включаючи Алана Тюрінга, створила машину Bombe.

Але шифр Enigma не єдиний «незламний» шифр тих часів. У Китаї, під час громадянської війни між Комуністичною партією і Ґоміньданом користувалися так

званим «пурпуровим» шифром, який за деякими даними був якщо не на одному рівні з Enigma, ба був навіть складнішим. Але його вдалося, хоч і з величезними труднощами, дешифрувати командою Жона Дзіньдженя. Але на його місце прийшов «чорний» шифр, криптографічна складність якого була на найвищому рівні. Його так і не вдалося дешифрувати. [3]

Комп'ютери та електронні дані

З епохою комп'ютерів і електронних даних (1950-1980-ті роки) почали з'являтися методи, спрямовані на захист електронних даних і закладувалися основи криптографії в електронному середовищі. Щоправда спочатку вона використовувалася переважно у військових цілях, але у 1970-х роках криптографія почала розвиватися у цивільній сфері завдяки кільком революційним відкриттям. Одним з них стала поява асиметричної криптографії. Дослідники Вітфілд Хеллман і Мартін Діффі представили метод обміну ключами Діффі-Хеллмана, що відбувався між сторонами, які раніше не контактували. Цей метод значно підвищив рівень захисту електронних даних. Це було революційним проривом, оскільки раніше обмін ключами завжди вимагав безпечного фізичного передавання. Алгоритм Диффі-Геллмана започаткував напрямок, який пізніше став основою для сучасних технологій, таких як захищене з'єднання через Інтернет (SSL/TLS).[4]

У 1977 році Рон Рівест, Аді Шамір і Леонард Адлеман розробляють RSA – один з найстаріших і найперших методів шифрування для безпечної передачі даних, який використовується і донині. RSA базується на складності факторизації великих чисел (розкладання числа на прості множники) і якщо попередньо не знати ключа шифрування, його стає надзвичайно важко зламати навіть для найпотужніших комп'ютерів. Він широко використовується для електронних підписів, захисту електронної пошти та забезпечує конфіденційність в онлайн комунікаціях. [5]

У тому ж 1977 році запроваджується перший криптографічний стандарт – DES (Data Encryption Standart). Був розроблений IBM і затверджений Національним інститутом стандартів і технологій США (NIST). DES використовував симетричний алгоритм шифрування з ключем довжиною 56 біт завдяки чому бувшвидким і

ефективним для тогочасних комп’ютерів. DES став першим стандартом, який широко застосовувався у фінансовій сфері для захисту транзакцій, але через збільшення обчислювальної потужності комп’ютерів почав вважатися застарілим. У 2001 році його було замінено більш надійним алгоритмом шифрування AES (Advances Encryption Standart), який використовує набагато довший ключ шифрування, який неможливо зламати сучасним обладнанням. [6]

Мережеві технології

З розвитком мережевих технологій (1980-ті – 2000-ні роки) з’явилася потреба у захисті Інтернету У цей період почали формуватися сучасні підходи до забезпечення безпеки даних у глобальних мережах.

На початку 1980-х років Інтернет, який виріс з мережі ARPANET, почав використовуватися не лише у військових та наукових цілях, а й у бізнесі. Це призвело до різкого зростання кількості підключених пристроїв і, відповідно, збільшення ризиків.

Саме у цей період виникли перші спроби мережевих атак. Одим з найважливіших став хробак Morpica (1988 рік), про який згадувалося вище, який знерухомив значну частину тогочасного інтернету. Цей випадок добре показав, якими вразливими є мережеві системи, і став поштовхом до створення нових методів захисту.

Одним з головних задач цього періоду стало надання безпечної передачі даних у мережі. Для цього були розроблені перші протоколи безпеки, які стали основою сучасного Інтернету:

- **SSL (Secure Sockets Layer)** – розроблений у 1994 році компанією Netscape, Запезпечував шифрування даних між клієнтом і сервером. Дозволив створювати безпечні вебсайти і транзакції, особливо у сфері електронної комерції. У 2000-х роках став основним для захисту вебсайтів.
- **IPSec (Internet Protocol Security)** – розроблений у середині 1990-х років. Дозволяв шифрувати і аутентифікувати мережеві пакети на рівні IP. Став основою для створення віртуальних приватних мереж (VPN), які

використовуються для безпечного підключення до корпоративних мереж.

- **Стандарт PGP (Pretty Good Privacy)** – розроблений у 1991 році, став важливим кроком для захисту електронної пошти. Забезпечував шифрування повідомлень, що гарантувало їхню конфіденційність і цілісність. [7]

З поширенням шкідливих програм у 1980-х роках почала розвиватися антивірусна індустрія. У 1987 році були створені перші комерційні антивірусні програми, такі як VirusScan (від McAfee) та NOD (від ESET). Ці програми дозволяли виявляти і видаляти шкідливі файли, захищаючи персональні комп'ютери та сервери. [8]

У 1990-х роках з розвитком локальних мереж виникла потреба у захисті корпоративних систем від зовнішніх загроз. Це привело до розробки перших міжмережевих екранів (Firewall або ж брандмауер). Таким чином у 1988 році компанія Digital Equipment Corporation створила перший комерційний фаєрвол, який відокремлював небезпечні зовнішні мережеві з'єднання (наприклад, Інтернет) від безпечних внутрішніх. І до середини 1990-х років фаєрволи стали стандартним інструментом для захисту мереж від несанкціонованого доступу. [9]

Але все ж таки однією з головних загроз 1990-х років стали розподілені атаки на відмову в обслуговуванні (DDoS). Такі атаки були спрямовані на виведення серверів з ладу через перевантаження їх численними запитами. У відповідь на ці атаки почали створювати технології захисту, зокрема створювали системи виявлення вторгнень (IDS), які аналізували трафік і виявляли підозрілу активність, та використовували резервні сервери для балансування навантаження і зменшення впливу DdoS-атак.

Сучасні методи захисту інформації та виклики в сфері кібербезпеки

У період з 2010-х років і до сьогодення розвиток методів захисту інформації був спричинений різким зростанням цифрових технологій, таких як хмарні сервіси, мобільні пристрої, штучний інтелект та Інтернет речей (IoT). Кіберзагрози стали набагато складнішими та цілеспрямованими, що вимагало переходу до багаторівневих підходів у безпеці та активного використання інтелектуальних

технологій.

Широке впровадження хмарних сервісів у бізнесі та державному секторі підвищило потребу у посиленій безпеці даних. Сьогодні дані у хмарі шифруються як під час передачі, так і при зберіганні, а спеціалізовані системи, такі як AWS GuardDuty, дозволяють моніторити мережі та виявляти загрози. Штучний інтелект почав активно застосовуватися для аналізу поведінки користувачів, виявлення аномалій та автоматичного і швидкого реагування на потенційні загрози. Системи на основі штучного інтелекту, такі як Splunk або IBM QRadar, забезпечують ефективний захист у реальному часі, надаючи змогу аналізувати великі обсяги даних.

Програми-вимагачі стали однією з найсерйозніших загроз останнього десятиліття. Відомі атаки, такі як WannaCry (згадувалося вище), завдали багатомільярдних збитків компаніям по всьому світу. Захист від таких загроз включає регулярне створення резервних копій даних, розділення мереж та використання сучасного антивірусного програмного забезпечення. Водночас розвиток квантових комп'ютерів поставив під сумнів ефективність традиційних алгоритмів шифрування, таких як RSA чи AES. Для протидії цьому розробляються квантово-стійкі алгоритми, зокрема CRYSTAL-Kyber, які можуть запобігти зламам, пов'язаним із квантовими обчисленнями. [10]

Останнім часом увага також приділяється вразливостям «нульового дня» − нерозкритих або невиявлених вразливостей у програмному забезпеченні і їхня кількість стрімко зростає. У 2023 році було зафіксовано 97 таких вразливостей, що значно більше порівняно з попередніми роками, а на 2024 рік, поки що, виявлено 87 критичних вразливостей. [11] Зловмисники використовують ці прогалини для одночасних атак на велику кількість організацій, що вимагає від бізнесу активного впровадження превентивних заходів безпеки. Штучний інтелект і автоматизація стали ключовими інструментами як для атаки, так і для захисту. Вони дозволяють не лише аналізувати поведінку користувачів, але й прогнозувати потенційні загрози.

Регуляторна сфера також активно розвивається. Наприклад, у 2018 році в Європейському Союзі був прийнятий регламент GDPR, який встановив суворі вимоги до обробки та захисту персональних даних. Паралельно з цим нові акти, такі як Cyber Resilience Act, висувають ще жорсткіші вимоги до бізнесу, змушуючи організації адаптувати свої стратегії безпеки до змінюваних умов. [12]

Таким чином, сучасні методи кіберзахисту поєднують новітні технології, такі як хмарна безпека, багатофакторна автентифікація, штучний інтелект і квантова криптографія, із жорсткими стандартами регулювання. Вони спрямовані на протидію все складнішим кіберзагрозам, забезпечуючи ефективний захист інформації у цифрову епоху.

Висновки до розділу 1

Проведене дослідження в рамках першого розділу дозволило проаналізувати історичний розвиток кіберзагроз та методів захисту інформації, що стало основою для розуміння сучасних викликів у сфері кібербезпеки. Було виявлено, що кіберзагрози, які виникли разом із появою перших комп’ютерних систем, еволюціонували до складних багатошарових атак із використанням штучного інтелекту та соціальної інженерії. Водночас розвиток методів захисту інформації пройшов шлях від античних шифрів до сучасних криптографічних стандартів, таких як AES та RSA.

Значення цього розділу полягає у висвітленні важливості систематичного підходу до вивчення кіберзагроз та аналізу історичних уроків для подальшого вдосконалення систем кіберзахисту. Особливий акцент зроблено на використанні багаторівневих рішень, таких як шифрування, багатофакторна автентифікація та інтегровані системи моніторингу, що стали базовими інструментами сучасної кібербезпеки. Ці результати є важливими для формування надійних стратегій захисту у відповідь на зростаючі ризики.

У перспективі подальші дослідження мають бути спрямовані на адаптацію систем захисту до нових викликів, таких як загрози з боку квантових обчислень, а також на впровадження інноваційних технологій, включаючи штучний інтелект та

машинне навчання, для запобігання невідомим атакам.

РОЗДІЛ 2
МЕТОДОЛОГІЧНІ ПІДХОДИ ТА КОНЦЕПЦІЇ КІБЕРБЕЗПЕКИ

2.1. Класифікація кіберзагроз і методи атак

У загальному кіберзагрози можна поділити на такі види:

- **Таргетовані атаки (advanced persistent threat)**. В загальному можна виділити два види атаки на комп'ютерні системи. Перший варіант – шкідливе програмне забезпечення(наприклад, вірус, троянський кінь або ж хробак), маючи за мету нашкодити якомога більшій кількості систем. Другий варіант – атакувати прицільно (звідки й така назва) для компрометації комп'ютерів конкретних користувачів (які мають справу з особливо цінною інформацією) або конкретної установи.
- **Кібертероризм.** Це можливість впливати через комп'ютерну мережу (локальну або Інтернет) на технологічні чи виробничі процеси. Інформаійно-комунікаційні технології надають терористам величезний перелік інструментів, що даються можливість готувати керувати та координувати теракти за допомогою комп'ютерних мереж, та втручатися практично у будь-який технологічний процес, що керується цифровою системою керування.
- **Кібервійни**. Stuxnet – дуже складний комп'ютерний хробак, який став відомим у 2010 році, є першою у світі масштабною кіберзброєю для ведення кібервійни. Він широко використовувався для диверсій або відключення систем. [13]
- **Хактивізм (сполучення слів хакінґ та активізм)** – це акт зловживання інформацією у соціальних мережах. Деякі хакерські угруповання ставлять собі головною метою видобування конфіденційної інформації і успішно ввикладають її в Інтернеті у вільний доступ. Це, зазвичай, стосується різного штибу скандалів, корупції, змов, а також розкриття незручних таємниць, які

суперечать закону та й будь-яким людським цінностям. Також вони це можуть робити для привертання уваги громадськості до того, що на їхню думку, є важливою проблемою чи причиною, як от свобода інформації, права людини чи релігійна точка зору. Відомими хактивістами є Anonymous і LulzSec. [14]

- **Атаки на банківські системи.** Досить поширене явище, особливо, після того, як у банківській сфері почали застосовувати інформаційно-комунікаційні технології. Нерідкими випадками є фішинг, викрадення даних платіжних карток, а також застосування шкідливого програмного забезпечення для втручання в роботу систем банку.
- **Атаки на електронний уряд.** «Електронний уряд» – це певна інформаційно-комунікаційна система, або їхнє об'єднання, що спрощує для держави надання державних послугав для громадян. Спрямовані атаки можуть сильно зашкодити функціонуванню усієї такої системи. [15]

За джерелами загроз виділяють:

- **Зовнішні загрози**: атаки хакерів, злочинних угруповань, державних структур або конкурентів.
- **Внутрішні загрози**: пов'язані з діями співробітників організації – навмисними (шпигунство, саботаж) або ненавмисними (помилки через недбалість чи низьку обізнаність).

За мотивами атак:

- **Фінансова вигода**: Викрадення даних для подальшого продажу, вимагання викупу, шахрайство.
- **Політичний вплив**: Кібератаки, що використовуються для шпигунства, маніпуляції громадською думкою або дискредитації конкурентів.
- **Хактивізм**: Атаки з метою досягнення соціальних або політичних змін.
- **Кібертероризм**: Масштабні атаки на критичну інфраструктуру з метою завдання значної шкоди.

За типами впливу на інформаційні системи:

- **Порушення конфіденційності**: Викрадення або розголошення даних.
- **Порушення цілісності**: Модифікація даних без дозволу.
- **Порушення доступності**: Блокування доступу до інформації або ресурсів (атаки на відмову в обслуговуванні, DDoS). [16]

Щодо методів атак. Що тільки кіберзлочинці не використовують для досягнення своїх цілей. До найпоширеніших методів можна віднести:

- **Шкідливе програмне забезпечення (Malware)**: Програми, які проникають у систему для викрадення, пошкодження або блокування інформації. Види:
  - **Віруси**: Самовідтворювані програми, що поширюються через файли.
  - **Трояни**: Програми, що маскуються під легітимні, але виконують шкідливі дії.
  - **Програми-вимагачі (Ransomware)**: Блокують доступ до даних і вимагають викуп.
- **Фішинг (Phishing)**: Використання електронних листів або вебсайтів, що імітують легітимні, для викрадення даних користувача (паролі, банківські дані).
- **Атаки на відмову в обслуговуванні (DoS/DDoS)**: Мережеві атаки, які перевантажують сервери запитами, що робить їх недоступними для користувачів.
- **Атаки методом грубої сили (Brute Force)**: Автоматизовані спроби підбору паролів шляхом перебору можливих комбінацій.
- **Соціальна інженерія**: Маніпуляція людьми з метою отримання доступу до конфіденційної інформації.
- **Атаки на ланцюг постачання (Supply Chain Attacks)**: Зловмисники атакують постачальників програмного забезпечення або апаратного забезпечення для отримання доступу до цільових систем. Відомий приклад – атака на SolarWinds у 2020 році.
- **Zero-Day атаки**: Використання вразливостей у програмному забезпеченні до

моменту їхнього виправлення розробниками.

- **Снифінг (Sniffing)**: Перехоплення даних, які передаються мережею, з метою їхнього аналізу або викрадення.
- **Експлойти**: Використання вразливостей у програмному забезпеченні для виконання шкідливих дій. [17]

2.2. Методи захисту інформації

Методи захисту інформації є важливою складовою забезпечення кібербезпеки у сучасному цифровому світі. Вони охоплюють широкий спектр технологій, включаючи системи автентифікації, шифрування, цифрові підписи та багаторівневі платформи захисту. Ці методи працюють у комплексі для забезпечення конфіденційності, цілісності й доступності даних, захищаючи їх як від зовнішніх, так і від внутрішніх загроз.

Системи захисту інформації є фундаментальною інфраструктурою для забезпечення безпеки даних. Вони використовуються для запобігання несанкціонованого доступу, виявлення загроз та їх нейтралізації. Основними елементами таких систем є:

- **Системи контролю доступу**: Обмежують права користувачів на доступ до даних беручи до уваги їхні ролі та повноваження. Моделі доступу, такі як DAC (Discretionary Access Control) або RBAC (Role-Based Access Control), дозволяють адмініструвати рівні доступу.
- **Системи моніторингу загроз**: Інструменти на кшталт SIEM (Security Information and Event Management) забезпечують аналіз активності в мережах та виявлення аномальної поведінки, яка може свідчити про атаку.
- **Резервне копіювання і відновлення**: Системи бекапу дозволяють мінімізувати втрати даних у разі успішної атаки, технічного збою або стихійного лиха.

- **Системи реагування на інциденти**: Включають механізми блокування підозрілої активності, ізоляції вражених систем і автоматичного повідомлення про загрози. [18]

Автентифікація є початковим етапом контролю доступу, оскільки вона дозволяє підтвердити особу користувача. У традиційних системах для цього використовуються паролі, однак сучасні системи дедалі частіше впроваджують дво- або багатофакторну автентифікацію. Це метод, що вимагає підтвердження через кілька факторів, таких як пароль, одноразовий код зі смартфона або біометричні дані, наприклад відбиток пальця чи сканування обличчя. Такий підхід значно знижує ризик несанкціонованого доступу навіть у випадку компрометації одного з факторів.

Авторизація, що йде після автентифікації, визначає, які ресурси користувач може використовувати та в якому обсязі, застосовуючи, наприклад, модель розподілу доступу на основі ролей (RBAC), або ж на основі атрибутів (ABAC). [19]

Шифрування є одним із найнадійніших методів захисту даних, який гарантує їхню конфіденційність навіть у разі перехоплення зловмисниками. У сучасних системах шифрування широко використовуються симетричні алгоритми, такі як AES (Advanced Encryption Standard), які забезпечують високу швидкість шифрування. Однак асиметричні алгоритми, наприклад RSA, забезпечують додаткову безпеку, використовуючи два ключі — публічний і приватний. Вони часто застосовуються у поєднанні для досягнення оптимальної продуктивності та рівня безпеки. Наприклад, у протоколах шифрування для вебсайтів (HTTPS) асиметричне шифрування використовується для встановлення з'єднання, після чого дані передаються за допомогою симетричного шифрування.

Цифрові підписи є важливим засобом забезпечення цілісності та автентичності даних. Вони дозволяють підтвердити, що дані не були змінені під час передачі, і що вони походять від довіреної особи. Використання цифрових підписів базується на асиметричній криптографії, де приватний ключ підписувача створює підпис, а публічний ключ дозволяє перевірити його справжність. Це часто використовується

для юридичних документів, контрактів, угод, а також для забезпечення безпеки програмного забезпечення. [20]

Усі ці методи інтегруються у більш складні системи захисту інформації, які забезпечують як захист даних, так і виявлення та запобігання загрозам. Наприклад, сучасні системи моніторингу, такі як SIEM, про які написано вище, аналізують активність у мережі та дозволяють ідентифікувати підозрілу поведінку. Також важливу роль відіграє резервне копіювання, яке дозволяє відновити дані у разі успішної атаки або технічного збою.

Сучасні системи захисту інформації інтегрують ці технології у багаторівневу екосистему безпеки. Наприклад, у корпоративних середовищах використовуються шифрування для передачі даних, багатофакторна автентифікація для входу до системи, а також цифрові підписи для забезпечення невідмовності. Цей комплексний підхід дозволяє знизити ризики, пов’язані з витоком інформації та кіберзагрозами.

Таким чином, ефективний захист інформації базується на поєднанні різних методів і систем. Автентифікація, шифрування, цифрові підписи та комплексні системи моніторингу є взаємодоповнюючими компонентами, що забезпечують безпеку інформації в умовах сучасного цифрового середовища.

2.3. Засоби виявлення та протидії кібератакам

Сучасні кібератаки стають дедалі складнішими, що вимагає впровадження ефективних засобів виявлення та протидії. Ці засоби спрямовані на знаходження, аналіз і знешкодження загроз у реальному часі, а також на запобігання їхньому повторенню.

Одним із ключових інструментів у боротьбі з кібератаками є:

- **Система виявлення IDS (Intrusion Detection System)**: призначена для аналізу мережевого трафіку та виявлення підозрілої активності, яка може свідчити про атаку. IDS зазвичай діє в пасивному режимі та повідомляє

адміністраторів про потенційні загрози.

- **Система запобігання IPS (Intrusion Prevention System)**: поєднує функції IDS із можливістю активного втручання. Вона автоматично блокує підозрілі запити або ізолює вразливі системи.

Детальніше про **систему управління подіями та інформацією безпеки (SIEM)**. Вона є дуже важливим елементом сучасного кіберзахисту. Вона дозволяють збирати, аналізувати та корелювати дані про події у мережі та системах для виявлення загроз.

Функції SIEM:

- Моніторинг мережевого трафіку та журналів подій.
- Аналіз поведінкових аномалій.
- Реалізація автоматичних дій для нейтралізації загроз (наприклад, блокування IP-адрес).

SIEM-системи дозволяють виявляти навіть складні атаки, які використовують декілька точок входу або відбуваються у кілька етапів. Приклади таких систем: Splunk, IBM QRadar, ArcSight.

Розширені системи детекції та реагування:

- **EDR (Endpoint Detection and Response)**: Орієнтована на кінцеві точки (комп'ютери, сервери, мобільні пристрої). EDR дозволяє виявляти та ізолювати шкідливу активність на рівні пристроїв. Приклади таких систем: CrowdStrike Falcon, Microsoft Defender for Endpoint.
- **XDR (Extended Detection and Response)**: Поєднує аналіз даних з мереж, кінцевих точок, серверів і хмарних платформ. XDR забезпечує ширше охоплення загроз і дозволяє виявляти складні багаторівневі атаки. [21]

Не забуваймо і про **антивірусні системи**. Вони залишаються базовим інструментом для виявлення та видалення шкідливих програм. Сучасні антивіруси використовують не лише бази сигнатур, але й технології машинного навчання для виявлення невідомого шкідливого ПЗ.

Фаєрволи, у свою чергу, контролюють вхідний і вихідний трафік у мережі, запобігаючи несанкціонованим з'єднанням. Вони вже були детальніше описані в пункті 1.3.4.

Також існують **системи виявлення аномалій на основі ШІ**. Штучний інтелект став ключовим інструментом у сучасних засобах детекції. Він дозволяє аналізувати великі обсяги даних у реальному часі, виявляючи невідомі загрози за аномальною поведінкою. Як приклад: виявлення фішингових електронних листів, виявлення незвичайного підключення до корпоративних мереж або аналіз поведінки користувачів для запобігання компрометації облікових записів.

Крім того існує також **протидія розподіленим атакам на відмову в обслуговуванні (DDoS)**, що залишаються однією з найсерйозніших загроз. Для їхньої протидії використовуються спеціалізовані платформи, такі як Cloudflare або Akamai, які можуть: виявляти та блокувати шкідливий трафік, розподіляти навантаження між серверами, забезпечувати безперервну роботу ресурсів навіть під час атаки.

Ну і найпростіше, що можна зробити задля протидії кібератак, так це прості **організаційні заходи**, як от навчання персоналу приніипам кібергігієни, або ж самостійне вивчення цього питання, а також проведення регулярних тестувань безпеки. [22]

Висновки до розділу 2

Другий розділ дав змогу глибше зрозуміти методологічні аспекти кібербезпеки та сучасні підходи до захисту інформації. Було проаналізовано класифікацію кіберзагроз, методи атак та засоби їхньої нейтралізації, що дозволило систематизувати знання про основні ризики в цифровому середовищі. Значна увага приділена інтеграції сучасних методів захисту, таких як системи виявлення вторгнень, шифрування та багатофакторна автентифікація, що підвищують загальний рівень безпеки.

Отримані результати мають важливе значення для формування ефективних політик кіберзахисту в умовах сучасного суспільства. Удосконалення систем

моніторингу та впровадження квантово-стійких алгоритмів шифрування дозволяють мінімізувати ризики, пов’язані із сучасними кіберзагрозами. Це створює фундамент для розвитку інноваційних технологій у сфері захисту даних.

Подальші дослідження можуть бути спрямовані на вдосконалення систем захисту критичної інфраструктури, адаптацію захисних механізмів до атак із використанням штучного інтелекту та забезпечення надійного захисту персональних даних у контексті міжнародних регуляторних стандартів.

РОЗДІЛ 3
ФОРМУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОЄКТУ

3.1. Мета, завдання проєкту та аналіз вимог

Метою даного проєкту є створення інтелектуальної системи діагностики на основі симптомів, яка дозволяє користувачам швидко отримувати можливі діагнози, аналізуючи введені симптоми. Система призначена для полегшення попереднього аналізу стану здоров'я, забезпечення персоналізованих рекомендацій щодо подальших дій, таких як консультація з лікарем, а також підвищення доступності базової діагностики для широкого загалу.

У систему інтегровано сучасні механізми кібербезпеки, які гарантують захищеність даних користувачів та мінімізують ризики компрометації системи. У проєкті використовується сучасний підхід до розробки веб-додатків, що дозволяє легко адаптувати систему до потреб різних категорій користувачів.

Система розробляється як веб-додаток із вбудованим чат-ботом, який у зручному інтерфейсі допомагає користувачам вводити симптоми, отримувати відповідні діагнози та рекомендації. Основними принципами, які закладені в основу системи, є доступність, простота використання, захищеність та гнучкість.

Для реалізації проєкту визначено кілька ключових завдань:

1. Розробка функціонального ядра системи:
    - Створення алгоритму, який аналізує введені симптоми та порівнює їх із базою даних діагнозів.
    - Впровадження механізму ваги симптомів для підвищення точності діагностики.
2. Інтеграція користувацького інтерфейсу;
3. Забезпечення високого рівня кібербезпеки:
    - Реалізація автентифікації користувачів через Google OAuth 2.0 для безпечного входу в систему.

- Захист від типових атак, таких як SQL-ін'єкції, CSRF, XSS.
- Шифрування збережених даних, таких як історія чатів, для запобігання несанкціонованому доступу.

4. Розробка бази даних:
    - Створення структури бази даних для зберігання симптомів, діагнозів, даних користувачів та історії чатів.
    - Оптимізація запитів до бази даних для забезпечення високої швидкодії навіть за значного навантаження.
5. Моніторинг і резервне копіювання:
    - Логування всіх дій користувачів, включаючи спроби входу, введення симптомів та очищення історії чатів, для моніторингу активності та забезпечення прозорості.
    - Автоматичне резервне копіювання бази даних для відновлення даних у разі аварійних ситуацій.
6. Забезпечення масштабованості та модульності: Побудова системи з можливістю додавання нових функцій, таких як інтеграція з медичними API, розширення бази симптомів та діагнозів, а також підключення додаткових модулів без значних змін у коді.

Аналіз вимог

1. Функціональні вимоги:
    - Користувачі повинні мати можливість вводити симптоми у текстове поле та отримувати список можливих діагнозів.
    - Система повинна зберігати історію чатів кожного користувача, дозволяючи її перегляд та очищення за запитом.
    - Інтерфейс має бути адаптивним, простим у використанні та забезпечувати швидку взаємодію.
2. Нефункціональні вимоги:
    - Високий рівень безпеки даних користувачів. Шифрування даних на

етапах передачі (SSL/TLS) та зберігання.

- Низький час відгуку для обробки запитів користувачів, незалежно від обсягу даних у базі.
- Доступність на різних пристроях і браузерах.

3. Технічні вимоги:
    - Використання Python і Flask для створення серверної частини.
    - PostgreSQL для зберігання даних.
    - HTML, CSS, JavaScript для побудови динамічного користувацького інтерфейсу.

Важливо також розуміти, що дана система не є заміною професійної медичної діагностики і не є абсолютно точною, а використовується лише як допоміжний інструмент. Діагнози базуються виключно на попередньо визначеному датасеті і не враховують унікальних характеристик користувача, таких як історія хвороб чи генетичні фактори.

3.2. Технології, компоненти та методи захисту в проєкті

3.2.1. Використані технології та обґрунтування їх вибору

У процесі розробки проєкту було обрано набір сучасних технологій і фреймворків, які забезпечують високу продуктивність, гнучкість, масштабованість та безпеку системи. Вибір технологій базувався на специфічних потребах проєкту, таких як інтеграція алгоритму діагностики, забезпечення зручного інтерфейсу для користувача та впровадження кіберзахисту.

Python було обрано як основну мову програмування для розробки серверної частини через його простоту, читабельність і величезну екосистему бібліотек. Python є ідеальним вибором для реалізації алгоритму діагностики та й усього проєкту загалом, завдяки його зручним бібліотекам. [23]

Flask використовується як веб-фреймворк для реалізації серверної логіки. Його перевагами є легковажність, модульність і простота інтеграції сторонніх бібліотек,

таких як Flask-WTF для CSRF-захисту або Flask-SQLAlchemy для роботи з базами даних. Flask також забезпечує гнучкість у налаштуванні маршрутизації та обробки запитів, що є критично важливим для побудови адаптивної системи діагностики. [24]

Бібліотека **Flask-SQLAlchemy** дозволяє взаємодіяти з базою даних на рівні Python-об’єктів, уникаючи використання сирих SQL-запитів. Це не тільки прискорює розробку, але й забезпечує захист від SQL-ін’єкцій. SQLAlchemy підтримує складні зв’язки між таблицями, що є важливим для зберігання даних користувачів, історії чатів і симптомів. [25]

PostgreSQL використовується як СУБД (система управління базами даних) її обрано завдяки її продуктивності, надійності та підтримці розширених функцій. PostgreSQL ідеально підходить для проєктів із великими обсягами даних, оскільки забезпечує ефективну обробку складних запитів і підтримує можливості шифрування та захисту даних. [26]

HTML5 та **CSS3** використовуються для розробки структури та стилізації веб-інтерфейсу. HTML5 забезпечує адаптивність сторінок, тоді як CSS3 дозволяє створювати сучасний дизайн із використанням анімацій, медіа-запитів та інших елементів.

JavaScript використовується для створення динамічного функціоналу, такого як динамічне розширення чи збільшення елементів або відображення результатів діагностики без перезавантаження сторінки. У поєднанні з AJAX JavaScript дозволяє реалізувати асинхронну взаємодію з сервером. [27]

Для спрощення роботи з DOM-елементами, обробки подій і виконання асинхронних запитів була обрана бібліотека **jQuery**. Вона також дозволяє зменшити обсяг коду, необхідного для реалізації основних дій на стороні клієнта. [28]

Для забезпечення CSRF-захисту у проєкті використовується **Flask-WTF**. Цей модуль автоматично генерує CSRF-токени для всіх форм і запобігає підробці запитів, захищаючи додаток від атак. [29]

Для обмеження частоти запитів і запобігання DDoS-атакам у проєкті інтегровано

Flask-Limiter. Це дозволяє уникнути перевантаження сервера та зловживання системою. [30]

Авторизація користувачів реалізована через **Google OAuth 2.0**, що забезпечує високий рівень безпеки та дозволяє користувачам входити в систему без створення окремого пароля. Цей метод значно спрощує процес входу, підвищуючи довіру до проєкту. [31]

Вибрані технології було обрано з урахуванням їхніх переваг, відповідності вимогам проєкту та здатності забезпечити високий рівень безпеки. Flask як легкий веб-фреймворк дозволяє швидко розробляти RESTful API та інтегрувати сторонні модулі для захисту, роботи з базою даних і управління сесіями. PostgreSQL забезпечує надійну обробку великих обсягів даних і можливість швидкого масштабування, що є важливим для проєктів із високими вимогами до продуктивності.

На стороні клієнта HTML5, CSS3 і JavaScript забезпечують динамічний та інтуїтивно зрозумілий інтерфейс, необхідний для ефективної взаємодії користувача з системою. Інтеграція з Google OAuth 2.0 спрощує процес входу та мінімізує ризики, пов’язані із зберіганням паролів.

Комбінація цих технологій забезпечує гнучкість, безпеку та масштабованість, роблячи систему зручною як для користувачів, так і для розробників. У наступному підпункті буде детально описано компоненти додатка, їхню взаємодію та архітектурну структуру, що дозволяє ефективно реалізувати всі етапи роботи системи.

3.2.2. Архітектурна структура та компоненти проєкту

Проєкт розроблений як багаторівнева веб-система, що базується на клієнт-серверній архітектурі. Основу складають три рівні:

Рівень клієнта (Frontend)

Рівень клієнта забезпечує інтерфейс взаємодії користувача із системою, реалізований за допомогою HTML, CSS і JavaScript, з акцентом на динамічність та інтерактивність.

Основні елементи:

- Чат-інтерфейс для введення симптомів і отримання діагнозів.
- Адаптивний текстовий інтерфейс для введення даних.
- Кнопки для очищення чату, входу та виходу з облікового запису.
- Інформаційний блок із рекомендаціями для користувачів.

Ключові компоненти:

- HTML-шаблони: Основний файл `index.html` містить структуру сторінки, включаючи текстове поле для введення симптомів, кнопку відправки, блок для виведення повідомлень чат-бота та випадаюче меню для управління функціоналом.
- CSS-файли: Забезпечують стильове оформлення додатка. Файл `styles.css` містить правила для адаптивності інтерфейсу, кольорову схему та стилізацію елементів.
- JavaScript: Додає динамічну поведінку інтерфейсу, включаючи:
  - Динамічне змінення висоти текстового поля залежно від кількості введених рядків.
  - Відправку повідомлення при натисканні Enter.
  - Очистку історії чату через відповідну кнопку.
  - Асинхронну взаємодію із сервером за допомогою fetch.

Рівень логіки застосунку (Backend)

Рівень логіки забезпечує обробку введених даних, виконання алгоритмів діагностики та взаємодію з базою даних. Реалізований на Python із використанням Flask як веб-фреймворку.

Основні завдання:

- Обробка введених користувачем симптомів.
- Запуск алгоритмів діагностики для порівняння симптомів із базою даних.
- Реалізація авторизації через Google OAuth 2.0.
- Взаємодія з базою даних для збереження історії чатів та отримання

інформації про діагнози.

Ключові компоненти:

- __init__.py: Ініціалізація додатка, підключення до бази даних, налаштування маршрутизації запитів і захисту.
- routes.py: Реалізація основних маршрутів, таких як обробка введених симптомів і авторизація.
- chatbot.py: Логіка роботи чат-бота, яка аналізує симптоми, виконує алгоритми діагностики та формує відповіді для користувача.
- models.py: Опис моделей бази даних.

Структуру усіх файлів проєкту зображена на рисунку 3.1.

```
root/
├── app/
│   ├── __init__.py        # Ініціалізація Flask-додатка, підключення бази даних та модулів
│   ├── routes.py          # Основні маршрути додатка (введення симптомів, очищення чату, авторизація)
│   ├── chatbot.py         # Логіка роботи чат-бота, обробка симптомів
│   ├── models.py          # Опис моделей бази даних (користувачі, історія чатів, симптоми, діагнози)
├── templates/
│   ├── index.html         # Інтерфейс для користувача
├── static/
│   ├── css/
│   │   ├── styles.css     # Стилі інтерфейсу
│   ├── js/
│   │   ├── script.js      # Логіка роботи інтерфейсу
├── config.py              # Конфігурація додатка (налаштування бази даних, ключі OAuth)
├── requirements.txt       # Залежності проєкту
├── run.py                 # Основний файл запуску додатка
```

Рисунок 3.1. Структура файлів проєкту

Рівень бази даних (Database)

Рівень бази даних зберігає структуровані дані про користувачів, симптоми, діагнози та історію чатів. Використовується PostgreSQL як система керування базою даних.

Основні таблиці:

- Users: Зберігає інформацію про користувачів, які авторизувалися через Google.
- ChatHistory: Історія чатів, включаючи введені симптоми та отримані діагнози.
- Diseases: Перелік захворювань із відповідними симптомами та їх вагами.

Архітектура бази даних оптимізована для швидкої обробки запитів і запобігання

SQL-ін’єкціям. Для збереження даних і захисту застосовуються інструменти SQLAlchemy та Flask-WTF.

Інтеграція компонентів

Архітектура проєкту побудована за принципом модульності, що дозволяє легко оновлювати та розширювати функціонал у майбутньому. Клієнтська частина відповідає за введення даних і відображення результатів, серверна — за обробку запитів і реалізацію логіки, а база даних — за зберігання інформації.

Завдяки інтеграції таких компонентів, як Google OAuth 2.0, Flask-Limiter і Flask-WTF, система забезпечує надійний захист даних користувачів та стійкість до атак. Додаток надає зручний інтерфейс, високу продуктивність і масштабованість, що робить його ефективним інструментом для діагностики на основі симптомів.

3.3. Реалізація основного функціоналу проєкту

3.3.1. Створення інтерфейсу користувача

Основна HTML-розмітка інтерфейсу розташована у файлі index.html (рис. 3.2).

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SymptomCheck</title>
    <link rel="stylesheet" href="static/css/styles.css">
</head>
<body>
    <header>
        <h1>SymptomCheck</h1>
        <nav>
            <div class="dropdownMenu">
                <button id="menuButton">Меню</button>
                <div id="dropdownContent">
                    <button id="clearChatButton">Очистити чат</button>
                    <button id="logoutButton">Вихід</button>
                </div>
            </div>
        </nav>
    </header>
    <main>
        <div id="chatbox">
            <div id="chatlogs"></div>
            <div id="chatinput">
                <textarea id="inputbox" placeholder="Введіть симптоми..." rows="1"></text
                <button id="sendbutton">Відправити</button>
            </div>
        </div>
    </main>
    <script src="static/js/script.js"></script>
</body>
</html>
```

Рисунок 3.2. HTML-розмітка

Вона відповідає за побудову таких ключових елементів:

- Шапка сторінки, де розташовона назва проєкту та кнопки взаємодії
- Блок чату, де відображаються усі введені повідомлення та відповіді чатбота
- Поле введення, яке відповідає за написання самих повідомлень з симптомами
- А також різні кнопки, які відповідають за надсилання, очищення історії чату, вхід або ж вихід із системи.

Використання тегів <header>, <main> забезпечує кращу читабельність коду. І всі елементи рохташовані таким чином, щоби коректно відображатися на різних пристроях та різних браузерах.

Для стилізації інтерфейсу використовується файл styles.css, розташований у папці static/css/. Тут стилізуються усі можливі елементи вебсторінки. (рис. 3.3)

```css
/* Загальні стилі */
body {
    margin: 0;
    font-family: Arial, sans-serif;
    background-color: #f4f4f9;
    color: #333;
    display: flex;
    flex-direction: column;
    align-items: center;
}

/* Заголовок */
header {
    width: 100%;
    padding: 10px 20px;
    background-color: #6200ea;
    color: white;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

header h1 {
    margin: 0;
    font-size: 24px;
}

/* Блок чату */
#chatbox {
    width: 80%;
    max-width: 800px;
    height: 60vh;
    display: flex;
    flex-direction: column;
    justify-content: flex-end;
    border: 2px solid #ddd;
    border-radius: 10px;
    background-color: white;
    overflow: hidden;
    margin: 20px 0;
}

#chatlogs {
    flex-grow: 1;
    padding: 10px;
    overflow-y: auto;
    border-bottom: 1px solid #ddd;
}

#chatinput {
    display: flex;
    align-items: center;
    padding: 10px;
}

#inputbox {
    flex-grow: 1;
    resize: none;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
    font-size: 14px;
}

#sendbutton {
    margin-left: 10px;
    padding: 10px 20px;
    border: none;
    background-color: #6200ea;
    color: white;
    font-size: 14px;
    cursor: pointer;
    border-radius: 5px;
}

#sendbutton:hover {
    background-color: #3700b3;
}

/* Випадаюче меню */
.dropdownMenu {
    position: relative;
    display: inline-block;
}

#dropdownContent {
    display: none;
    position: absolute;
    right: 0;
    background-color: white;
    border: 1px solid #ddd;
    box-shadow: 0px 4px 6px rgba(0, 0
    z-index: 1;
}

#dropdownContent button {
    padding: 10px;
    width: 100%;
    border: none;
    background-color: white;
    cursor: pointer;
    text-align: left;
}

#dropdownContent button:hover {
    background-color: #f4f4f9;
}

#menuButton:hover + #dropdownContent,
#dropdownContent:hover {
    display: block;
}
```

Рисунок 3.3. Стилі CSS

А JavaScript у файлі script.js, що розташований в папці static/js/, додає

інтерактивність. (рис. 3.4) Зокрема:

- Надсилання повідомлень і динамічне відображення чату.
- Додавання нового рядка при Shift + Enter.
- Очищення історії чату.
- Розкриття та приховування випадаючого меню.

```
document.getElementById("sendbutton").addEventListener("click", function () {
    const inputbox = document.getElementById("inputbox");
    const chatlogs = document.getElementById("chatlogs");

    const userMessage = inputbox.value.trim();
    if (userMessage !== "") {
        const userBubble = document.createElement("div");
        userBubble.textContent = userMessage;
        userBubble.className = "user-message";
        chatlogs.appendChild(userBubble);

        inputbox.value = "";
        chatlogs.scrollTop = chatlogs.scrollHeight;

        // Імітація відповіді від чат-бота
        setTimeout(() => {
            const botBubble = document.createElement("div");
            botBubble.textContent = "Бот: Аналізую ваші симптоми...";
            botBubble.className = "bot-message";
            chatlogs.appendChild(botBubble);
            chatlogs.scrollTop = chatlogs.scrollHeight;
        }, 1000);
    }
});

// Очищення чату
document.getElementById("clearChatButton").addEventListener("click", function () {
    document.getElementById("chatlogs").innerHTML = "";
});

// Випадаюче меню
document.getElementById("menuButton").addEventListener("click", function () {
    const dropdown = document.getElementById("dropdownContent");
    dropdown.style.display = dropdown.style.display === "block" ? "none" : "block";
});
```

Рисунок 3.4. Скрипти script.js

Але це далеко не все, на що спроможний js, бо також за допомогою нього були реалізовані й інші функції, наприклад, авторизація, вхід і вихід з облікового запису, перевірка логіну, завантаження з бази даних історії чату тощо. Ці елементи та іхня робота детально описані в наступних пунктах.

А на рисунку 3.5 показана готова веб-сторінка, з якою ми і будемо працювати весь наступний час.

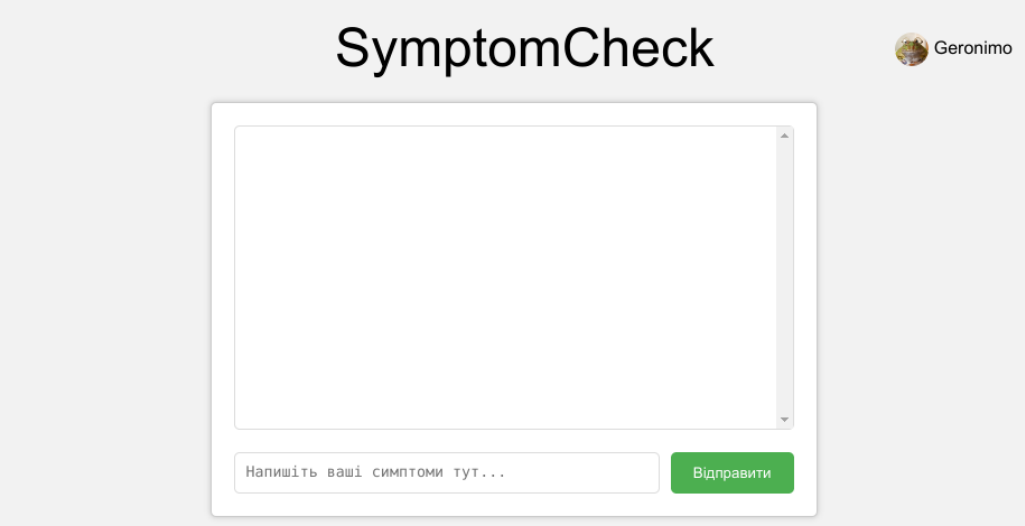

Рисунок 3.5. Готова веб-сторінка

Реалізація інтерфейсу користувача поєднує HTML-шаблони для структури, CSS для оформлення та JS для динамічного функціоналу. Розташування CSS і JS в окремих файлах забезпечує модульність та зручність оновлення коду.

3.3.2. Робота з базою даних

Даний проєкт був розроблений на операційній системі Linux з дистрибутивом Pop!_OS і встановлення PostgreSQL тут відрізняється від звичного встановлення на Windows. Як і для встановлення майже кожної програми чи утиліти на Linux треба використовувати термінал та спеціальні команди. Усі команди необхідні для встановлення показані на рисунку 3.6.

```
sudo apt update
sudo apt install postgresql postgresql-contrib
sudo systemctl start postgresql
sudo systemctl enable postgresql
sudo -i -u postgres
psql
CREATE DATABASE db_name;
CREATE USER db_user WITH PASSWORD 'db_password';
GRANT ALL PRIVILEGES ON DATABASE db_name TO db_user;
\q
psql -U db_user -d db_name -h localhost -W
export DATABASE_URL="postgresql://db_user:db_password@localhost/db_name"
source ~/.bashrc
flask db init
flask db migrate -m "Initial migration"
flask db upgrade
```

Рисунок 3.6. Команди для встановлення PostgreSQL

Структура бази даних складається з кількох таблиць, визначених у файлі

models.py. (рис. 3.7)

```python
from app import db  # Імпорт з app безпосередньо
from flask_login import UserMixin

# Модель користувача
class User(db.Model, UserMixin):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(150), unique=True, nullable=True)
    email = db.Column(db.String(150), unique=True, nullable=False)
    google_id = db.Column(db.String(200), unique=True, nullable=True)
    access_token = db.Column(db.String(500), nullable=True)
    avatar_url = db.Column(db.String(500), nullable=True)

    def __repr__(self):
        return f"<User {self.email}>"

# Модель захворювань
class Disease(db.Model):
    __tablename__ = 'diseases'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), nullable=False)
    symptoms = db.Column(db.String(500), nullable=False)

# Модель історії чату
class ChatHistory(db.Model):
    __tablename__ = 'chat_history'
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    message = db.Column(db.Text, nullable=False)
    diagnosis = db.Column(db.String(2000), nullable=False)
    timestamp = db.Column(db.DateTime, default=db.func.current_timestamp())

    user = db.relationship('User', backref=db.backref('chat_history', lazy=True))

    def __repr__(self):
        return f"<ChatHistory user_id={self.user_id} diagnosis={self.diagnosis}>"
```

Рисунок 3.7. Структура баази даних

User зберігає дані користувачів (ім'я, email, токен авторизації).

ChatHistory містить історію чатів, включаючи введені симптоми та отримані діагнози.

Diseases зберігають список захворювань і симптомів із вагами.

Для підключення до самої бази даних, до PostgreSQL, використовується файл конфігурації config.py (рис. 3.8), в якому зберігаються усі важливі дані та чутлива інформація. Для прикладу будемо використовувати простий файл, але на реальних проєктах краще розміщувати усі чутливі дані в змінних середовищах.

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager
from app.models import User

app = Flask(__name__)
app.config.from_object('config.Config')

db = SQLAlchemy(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login'

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

Рисунок 3.8. Підключення до PostgreSQL

А для ініціалізації бази даних нам знадобиться файл __init__.py. Крім того тут ще й також відбувається ініціалізація самого серверу, підтягування файлу конфігурації та завантаження самого користувача.

Як відбувається взаємодія з базою даних? Для збереження даних у базу існує функція diagnose (рис. 3.9) з файлу routes.py, яка зберігая всю історію чату авторизованого користувача.

```python
@app.route('/diagnose', methods=['POST'])
def diagnose():
    symptoms = request.form.get('symptoms')
    cleaned_symptoms = bleach.clean(symptoms)
    diagnosis = get_diagnosis(cleaned_symptoms)
    app.logger.info(f"Symptoms submitted: {cleaned_symptoms}")
    app.logger.info(f"Diagnosis generated: {diagnosis}")

    # Перевірка, чи користувач залогінений
    if current_user.is_authenticated:
        user_id = current_user.id
        app.logger.info(f"User ID: {user_id} is authenticated, saving data...")
        chat_record = ChatHistory(user_id=user_id, message=cleaned_symptoms, diagnosis=diagnosis)
        try:
            db.session.add(chat_record)
            db.session.commit()
            app.logger.info(f"Data saved for user ID: {user_id}")
        except Exception as e:
            app.logger.error(f"Error saving data for user ID: {user_id}: {e}")
            db.session.rollback()
    else:
        app.logger.warning("User is not authenticated, data not saved.")

    return jsonify({'diagnosis': diagnosis})
```

Рисунок 3.9. Функція dignose

Вона також перевіряє чи є користувач авторизований, а також вмикає логування.

А наприклад для очищення чату, тобто для видалення історії повідомлень використовується функція clear_chat (рис. 3.10).

```python
@app.route('/clear_chat', methods=['POST'])
@login_required
def clear_chat():
    try:
        # Видалення чату з бази даних
        ChatHistory.query.filter_by(user_id=current_user.id).delete()
        db.session.commit()
        return jsonify({"message": "Чат очищено"}), 200
    except Exception as e:
        db.session.rollback()
        app.logger.error(f"Помилка при очищенні чату для користувача {current_user.id}: {e}")
        return jsonify({"message": "Помилка при очищенні чату"}), 500
```

Рисунок 3.10. Функція очищення чату

Також існують й інші функції, зокрема get_history, яка завантажує історію повідомлень користовача та виводить її в чаті, user_info, яка повертає ім’я користувача, його аватарку, електронну пошту тощо

Таким чином PostgreSQL можна використовувати як надійне та швидке сховище даних, яке легко налаштувати

3.3.3. Взаємодія клієнт-сервер

У проєкті реалізована взаємодія клієнтської та серверної частин через асинхронні запити, що забезпечує ефективний обмін даними між користувачем і сервером. Основними компонентами цієї взаємодії є, як вже можна здогадатися є клієнтська частина (HTML, CSS, JavaScript), що відповідає за отримання введення від користувача, надсилання його на сервер і відображення відповідей, і сервверна частина (Flask), яка обробляє запити, виконує логіку діагностики та повертає результати

Клієнтська частина відповідає за динамічну взаємодію з користувачем і надсиланням запитів на сервер. Вона розташована у файлі script.js (рис 3.11).

Через поле textarea користувач вводить симптоми. При натисканні кнопки "Відправити" дані передаються у вигляді POST-запиту на сервер. Отримані від сервера результати додаються в чат як повідомлення бота. Якщо сталася помилка, користувач отримує відповідне повідомлення. Використання fetch дозволяє не перезавантажувати сторінку після кожного запиту.

```
$document.getElementById("sendbutton").addEventListener("click", function () {
    const inputbox = document.getElementById("inputbox");
    const userMessage = inputbox.value.trim();

    if (userMessage !== "") {
        // Відображення повідомлення користувача у чаті
        const chatlogs = document.getElementById("chatlogs");
        const userBubble = document.createElement("div");
        userBubble.textContent = userMessage;
        userBubble.className = "user-message";
        chatlogs.appendChild(userBubble);
        inputbox.value = "";
        chatlogs.scrollTop = chatlogs.scrollHeight;

        // Надсилання даних на сервер
        fetch("/diagnose", {
            method: "POST",
            headers: {
                "Content-Type": "application/x-www-form-urlencoded",
            },
            body: `symptoms=${encodeURIComponent(userMessage)}`,
        })
            .then((response) => response.json())
            .then((data) => {
                // Відображення відповіді сервера
                const botBubble = document.createElement("div");
                botBubble.textContent = `Бот: ${data.diagnosis}`;
                botBubble.className = "bot-message";
                chatlogs.appendChild(botBubble);
                chatlogs.scrollTop = chatlogs.scrollHeight;
            })
            .catch((error) => {
                console.error("Помилка:", error);
                const errorBubble = document.createElement("div");
                errorBubble.textContent = "Бот: Сталася помилка. Спробуйте ще раз.";
                errorBubble.className = "error-message";
                chatlogs.appendChild(errorBubble);
            });
    }
});
```

Рисунок 3.11. Код для надсилання повідомлень

Серверна частина обробляє запити клієнта і виконує логіку додатка. Вона реалізована у функції diagnose (рис. 3.9).

Сервер приймає дані у форматі application/x-www-form-urlencoded. Перевіряється, чи запит містить симптоми. Якщо ні, повертається HTTP-статус 400 із повідомленням про помилку. Для обробки даних викликається функція get_diagnosis, яка обробляє симптоми й повертає список можливих діагнозів (про неї детальніше розписано в наступному пункті 3.3.4). А формат відповіді як результат повертається у форматі JSON, що легко обробляється клієнтською частиною.

Взаємодія клієнт-сервер побудована таким чином, щоб забезпечити швидкість і надійність обміну даними. Клієнтська частина відповідає за асинхронну відправку запитів і відображення відповідей, тоді як сервер обробляє дані та виконує логіку.

Такий підхід гарантує зручність для користувача та масштабованість системи.

3.3.4. Реалізація алгоритму діагностики

Алгоритм діагностики побудований на основі аналізу введених користувачем симптомів, їх порівнянні з базою даних захворювань та використанні ваг симптомів для оцінки їхньої значущості. Це забезпечує підвищену точність результатів, оскільки враховується як частота зустрічання, так і унікальність кожного симптому. Розглянемо детально кожен етап роботи алгоритму з прикладами.

Етап 1. Введення симптомів користувачем. Користувач вводить перелік симптомів через текстове поле. Наприклад, він вводить:

«кашель, утруднене дихання, втома»

Система приймає ці дані й переходить до попередньої обробки. Тут важливо забезпечити зручність введення, тому користувач може вводити симптоми як списком через коми, пробіли, суцільним текстом, та й з додатковими словами (наприклад, я, маю, у мене, відчуваю, виникає, дуже тощо). Також система розуміє, що можуть бути помилки нписання, або ж різні слова, які мають одне й те саме значення.

Етап 2. Попередня обробка даних. На цьому етапі введений текст очищується. Видаляються зайві пробіли, текст переводиться в нижній регістр для нормалізації, видаляються усі непотрібні системі слова, а симптоми розділяються на окремі елементи. У результаті система отримує такий список:

["кашель", "утруднене дихання", "втома"]

Дублікати усуваються, якщо вони були, а неприйнятні символи ігноруються. Наприклад, якщо користувач помилково ввів "кашель, кашель", результатом буде унікальний список симптомів без повторів.

Етап 3. Порівняння із датасетом. Список очищених симптомів порівнюється з базою захворювань. У базі кожне захворювання має список своїх характерних симптомів із відповідними вагами. Наприклад, для грипу характерні симптоми можуть мати такі теоретичні ваги:

- кашель: 2

- утруднене дихання: 1
- біль у горлі: 2
- біль у грудях: 1
- лихоманка: 3
- втома: 1

Для астми:

- кашель: 2
- утруднене дихання: 3
- свистяче дихання: 4

Для пневмонії:

- кашель: 3
- утруднене дихання: 2
- біль у горлі: 1
- біль у грудях: 4
- втома: 1

Кожен симптом із введеного списку зіставляється з симптомами захворювань у базі, і його вага додається до загальної суми, якщо знайдено збіг.

Етап 4. Розрахунок ваги для кожного діагнозу. Після порівняння ваги симптомів підсумовуються для кожного можливого діагнозу. У нашому прикладі алгоритм обчислює такі суми:

- Грип: кашель (2) + утруднене дихання (1) + втома (1) = 4
- Астма: кашель (2) + утруднене дихання (3) = 5
- Пневмонія: кашель (3) + утруднене дихання (2) + втома (1) = 6

Результати сортуються у порядку зменшення загальної ваги.

Етап 5. Виведення результатів. Система вибирає топ-3 діагнози з найвищими вагами й виводить їх користувачу. У нашому прикладі користувач отримає:

1. Пневмонія: 6 балів. Рекомендація: зверніться до лікаря, можливе рентгенологічне обстеження.

2. Астма: 5 балів. Рекомендація: перевірте наявність тригерів, наприклад, алергенів.
3. Грип: 4 бали. Рекомендація: зверніться до терапевта для підтвердження діагнозу.

Якщо введені симптоми не відповідають жодному захворюванню, користувач отримує приблизно отаке повідомлення: «Вибачте, діагноз не може бути встановлений на основі введених даних. Уточніть симптоми або зверніться до лікаря».

Приклади роботи алгоритму в складніших випадках. Якщо користувач вводить лише один загальний симптом, наприклад, "кашель", система може видати кілька можливих діагнозів з низькою впевненістю, наприклад: грип (вага 2), пневмонія (вага 3), бронхіт (вага 2). У такому випадку система порекомендує уточнити симптоми. А при введенні дуже специфічного симптому, наприклад, "жовтяниця", алгоритм швидко визначає вузький перелік захворювань, таких як гепатит чи жовчнокам'яна хвороба, оскільки цей симптом має високу вагу для обох діагнозів.

Алгоритм дозволяє враховувати як загальні, так і специфічні симптоми, забезпечуючи більш точний аналіз, ніж просте порівняння списків. Наприклад, грип і пневмонія мають багато схожих симптомів, однак система з вагами може враховувати, що "біль у грудях" має вищу вагу для пневмонії, ніж для грипу.

Окрім того, такий підхід легко розширюється. У разі доповнення бази новими захворюваннями достатньо лише додати їхні симптоми із відповідними вагами, не змінюючи логіки роботи. Однак, попри свою ефективність, модель залежить від якості бази даних. Якщо в базі недостатньо даних або ваги симптомів визначені неправильно, це може вплинути на результати. Наприклад, якщо "лихоманка" має однакову вагу для грипу та малярії, системі буде важко розрізнити ці захворювання.

Алгоритм також не враховує контекстуальні дані, такі як вік, стать або хронічні захворювання користувача. Це робить його корисним як допоміжний інструмент, але не заміною професійної медичної діагностики.

Натомість загальний алгоритм аналізу симптомів має таку структуру:

1. Отримання симптомів від користувача.
2. Обробка симптомів для підготовки до аналізу.
3. Порівняння симптомів із базою даних захворювань.
4. Обчислення рівня впевненості для кожного захворювання.
5. Повернення результатів користувачу.

Отримання симптомів відбувається у клієнтській частині, у файлі script.js, а для обробки запиту використовується серверна частина, а саме функція diagnose у файлі routes.py, яка обробляє запит від клієнта, викликає алгоритм діагностики та повертає результати.

А ось основний алгоритм діагностики з вагами розташований у файлі chatbot.py, у функції get_diagnosis (рис. 3.12).

```
def get_diagnosis(user_input):
    user_symptoms = [clean_symptoms(symptom.strip().lower()) for symptom in user_input.split(',')]

    # Ініціалізація словника для збереження ваг збігів
    diseases_ratings = {disease: {'matched_weight': 0, 'total_weight': sum(symptoms.values())}
                        for disease, symptoms in diseases.items()}

    # Порівняння симптомів користувача з симптомами в базі даних
    for user_symptom in user_symptoms:
        for disease, symptoms in diseases.items():
            if user_symptom in symptoms:
                diseases_ratings[disease]['matched_weight'] += symptoms[user_symptom]

    # Обчислення рівня впевненості для кожного захворювання
    diagnoses_with_confidence = []
    for disease, ratings in diseases_ratings.items():
        matched_weight = ratings['matched_weight']
        total_weight = ratings['total_weight']
        if matched_weight > 0:
            confidence = round((matched_weight / total_weight) * 100, 2)  # У відсотках
            diagnoses_with_confidence.append({'disease': disease, 'confidence': confidence})

    # Сортування діагнозів за рівнем впевненості
    diagnoses_with_confidence.sort(key=lambda x: x['confidence'], reverse=True)

    # Вибір топ-3 діагнозів
    top_diagnoses = diagnoses_with_confidence[:3]

    # Формування результату для виведення
    if top_diagnoses:
        result = ', '.join([f"{d['disease']} ({d['confidence']}%)" for d in top_diagnoses])
    else:
        result = '''Вибачте, діагноз не може бути встановлений
                    на основі введених даних. Уточніть симптоми
                    та зверніться до лікаря'''

    return result
```

Рисунок 3.12. Функція get_diagnosis

Після виконання алгоритму усі результати (окрім помилок) зберігаються в базу даних в таблицю ChatHistory. І разом з цим виводяться результати у вигляді списку

діагнозів та хвороб на екран в чат і відображаються в клієнтській частині за допомогою fetch.

3.3.5. Реалізація системи авторизації

У цій системі авторизація реалізована через Google Oauth 2.0. А саме за допомогою бібліотеки Flask-Dance, яка спрощує інтеграцію OAuth із Google. [32]

Загальна логіка авторизації нашого проєкту складається з наступних етапів:

1. Запит авторизації. Користувач перенаправляється на сторінку Google для входу.
2. Отримання токенів. Після успішного входу Google повертає токени.
3. Перевірка профілю. Система отримує інформацію про користувача (ім'я, email, аватар, та й будь-яку іншу доступну).
4. Збереження даних. Якщо користувач новий, його інформація додається до бази даних.
5. Авторизація. Користувач залишається авторизованим завдяки сесії.

Для налаштування Oauth 2.0 у файл конфігурації (або ж у змінні середовища) потрібно додати так звані client_id та client_secret – унікальні ідентифікатори, які використовують під час інтеграції і дозволять сервісу ідентифікувати наш додаток.

Client Id – це публічний ідентифікатор нашого додатка, який повідомляється сторонньому сервісу (Google). Він використовується для визначення, який саме додаток робить запит і допомагає сервісу знати, ддо якого саме додатка прив’язувати дії користувача. Виглядає приблизно отак:

1234567890-abcdefg.apps.googleusercontent.com

Client Secret – це приватний ключ нашого додатка, який використовується для підтвердження його автентичності. Він має залишатися конфіденційним і зберігатися на сервері. Використовується разом із Client ID, щоб авторизувати запити додатка до сервісу. Має приблизно такий вигляд:

X1Y2Z3-A4B5C6D7_E8F9G0

Після успішної авторизації Google повертає код авторизації. Наш сервер використовує цей код разом з client_id і client_secret, щоб отримати Access Token.

Налаштування самої авторизації виконується у файлі __init__.py (рис. 3.13).

```python
# Налаштування Google OAuth
google_bp = make_google_blueprint(
    client_id=Config.GOOGLE_CLIENT_ID,
    client_secret=Config.GOOGLE_CLIENT_SECRET,
    scope=["openid", "https://www.googleapis.com/auth/userinfo.profile",
                     "https://www.googleapis.com/auth/userinfo.email"],
    redirect_to="google_logged_in"
)
app.register_blueprint(google_bp, url_prefix='/login')

# Завантаження користувача за його ID
@login_manager.user_loader
def load_user(user_id):
    from app.models import User
    return User.query.get(int(user_id))
```

Рисунок 3.13. Налаштування авторизації

Вхід та вихід відбувається у функціях login_with_google та logout (рис. 3.14).

```python
@app.route("/login/google")
def login_with_google():
    if current_user.is_authenticated and current_user.access_token:
        # Використовуємо збережений токен
        google_bp.session.token = {"access_token": current_user.access_token}
        return redirect(url_for("home"))  # Перенаправляємо користувача на головну

    return redirect(url_for("google.login"))

@app.route("/logout", methods=["POST"])
def logout():
    if current_user.is_authenticated:
        username = current_user.username
        user_id = current_user.id
        logout_user()
        # Логування спроби виходу
        app.logger.info(f"User {username} (ID: {user_id}) logged out.")
        return """
        <script>
            window.close();
        </script>
        """
    else:
        app.logger.warning("Attempted logout without an authenticated session.")
        return """
        <script>
            window.close();
        </script>
        """
```

Рисунок 3.14. Налаштування входу та виходу

При успішних спробах авторизації на сервер буде відправлятися відповідь від Google і обробляється вона за допомогою функції google_logged_in, що знаходиться у файлі __init__.py (рис. 3.15).

```python
@oauth_authorized.connect_via(google_bp)
def google_logged_in(blueprint, token):
    from app.models import User
    resp = blueprint.session.get("/oauth2/v1/userinfo")
    if resp.ok:
        account_info = resp.json()
        google_id = account_info["id"]
        email = account_info.get("email")
        username = account_info.get("name")
        avatar_url = account_info.get("picture")

        # Перевірка користувача
        user = User.query.filter_by(google_id=google_id).first()
        if user:
            user.access_token = token['access_token']
            user.username = username
            user.avatar_url = avatar_url
        else:
            user = User(google_id=google_id, email=email, access_token=token['access_token'],
                username=username, avatar_url=avatar_url)
            db.session.add(user)

        db.session.commit()

        # Авторизація користувача
        login_user(user, remember=True)
```

Рисунок 3.15. Відправка запиту до Google

За допомогою цієї функції отримуються токени доступу та інформація про користувача і якщо користувач новий, його дані додаються в базу. Модель користувача User вже була описана у пункті 3.3.2.

Для забезпечення захисту та легкого моніторингу реалізовано логування, яке записує усі спроби входу та виходу.

Таким чином авторизація через Google Oauth 2.0 це дуже надійний спосіб входу, що надає безпечне зберігання даних користувача завдяки зручності інтеграції з обліковим записом Google.

3.3.6. Розробка функціоналу чату

Клієнтська частина. Для нагадування знову покажемо HTML структуру чату (рис. 3.16), а повна HTML структура розписана у пункті 3.3.1.

```html
<div id="chatbox">
    <div id="chatlogs">
        <!-- Діалоги користувача та асистента -->
    </div>
    <div id="chatinput">
        <textarea id="inputbox" placeholder="Напишіть ваші симптоми тут..." row
        <button id="sendbutton" onclick="sendMessage()">Відправити</button>
    </div>
</div>
```

Усі повідомлення виводяться в chatlogs. Текст вводиться через inputbox, а відправляється за допомогою кнопки sendbutton.

Завантаження історії чату відбувається лише після успішної авторизації, за допомогою функції loadChatHistory (рис. 3.17).

```
function loadChatHistory() {
    $.get("/get_history", function(data) {
        $("#chatlogs").empty();  // Очищення старих записів
        data.forEach(record => {
            $("#chatlogs").append('<p class="user">' + record.message + '</p>');
            $("#chatlogs").append('<p class="bot">' + record.diagnosis + '</p>');
        });
        document.getElementById("chatlogs").scrollTop = document.getElementById("chatlogs").
    });
}
```

Очищення чату відбувається за допомогою спеціальної кнопки, що знаходиться в меню користувача і відгукується на функцію clearChat (рис. 3.18).

```
function clearChat() {
    const csrfToken = document.querySelector('meta[name="csrf-token"]').getAttribute('co

    fetch('/clear_chat', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'X-CSRFToken': csrfToken // Додаємо CSRF-токен
        }
    })
    .then(response => {
        if (response.ok) {
            alert("Чат очищено.");
            document.getElementById('chatlogs').innerHTML = ''; // Очищення чату на кліє
        } else {
            alert("Помилка при очищенні чату.");
        }
    })
    .catch(error => {
        console.error('Помилка:', error);
        alert("Щось пішло не так.");
    });
}
```

А найголовніша частина, а саме відправка повідомлень реалізується за допомогою функції sendMessage (рис. 3.19).

Серверна частина. Функціонал на сервері пов’язаний із маршрутами у файлі

```
// Функція для відправки повідомлення
function sendMessage() {
    var userMessage = $("#inputbox").val().trim();
    if (userMessage !== "") {
        // Додаємо повідомлення користувача в чат
        $("#chatlogs").append('<p class="user">' + userMessage + '</p>');

        // Відправка даних на сервер через AJAX
        $.ajax({
            type: "POST",
            url: "/diagnose",
            data: { symptoms: userMessage },
            headers: {
                "X-CSRFToken": csrfToken  // Додаємо CSRF-токен до заголовків запиту
            },
            success: function(response) {
                if (response.diagnosis == "Вибачте, діагноз не може бути встановлений на основі введених даних. Уточніть симптоми або зве
                    $("#chatlogs").append('<p class="bot">' + response.diagnosis + '</p>');
                else $("#chatlogs").append('<p class="bot">Ваші можливі діагнози: ' + response.diagnosis + '</p>');
                var chatlogs = document.getElementById("chatlogs");
                chatlogs.scrollTop = chatlogs.scrollHeight; // Прокрутка вниз до останнього повідомлення
            },
            error: function(error) {
                console.log(error);
            }
        });
        $("#inputbox").val(""); // Очищення поля введення після відправки
    }
}
```

routes.py.

Отримання історії чату відбувається за допомогою функції get_history (рис. 3.20).

```
@app.route('/get_history', methods=['GET'])
def get_history():
    if current_user.is_authenticated:
        user_id = current_user.id
        # Отримуємо всі повідомлення і діагнози користувача, впорядковані за часом
        history = ChatHistory.query.filter_by(user_id=user_id).order_by(ChatHistory.timesta
        # Перетворюємо дані в формат JSON для відправки на клієнт
        return jsonify([
            {
                "message": record.message,
                "diagnosis": record.diagnosis,
                "timestamp": record.timestamp
            } for record in history
        ])
    return jsonify([])  # Повертаємо порожній список, якщо користувач не залогінений
```

Рисунок 3.20. Функція завантаження історії чату на сервері

А от власне минула функція очищення чату не здатна власноруч очистити чат. Вона тільки може видалити повідомлення на стороні клієнта, а для видалення зі сторони серверу вона викликає функцію clear_chat, яка й видаляє історію повідомлень користувача (рис. 3.21).

А також для збереження нових повідомлень використовується функція diagnose, яка була описана в пункті 3.3.4.

А модель чату ChatHistory, у якій і зберігається уся історія листування, усі

```python
@app.route('/clear_chat', methods=['POST'])
@login_required
def clear_chat():
    try:
        # Видалення чату з бази даних
        ChatHistory.query.filter_by(user_id=current_user.id).delete()
        db.session.commit()
        return jsonify({"message": "Чат очищено"}), 200
    except Exception as e:
        db.session.rollback()
        app.logger.error(f"Помилка при очищенні чату для користувача {current_user.id}:
        return jsonify({"message": "Помилка при очищенні чату"}), 500
```

Рисунок 3.21. Функція очищення чату на сервері

повідомлення та відповіді, детальніше було розібрано в пункті 3.3.2.

3.4. Захист системи

Автентифікації і авторизація.

У нашій системі реалізована аутентифікація через Google OAuth 2.0, що гарантує безпеку його даних, оскільки все відбувається на сторонньому сервісі.

Також реалізованио захист сесій. Сесія автоматично завершується після виходу користувача. Але у нас також установлено обмеження часу (рис. 3.22), після якого сесія завершується і потрібен повторний вхід.

```python
from datetime import timedelta
app.config['REMEMBER_COOKIE_DURATION'] = timedelta(minutes=30)
```

Рисунок 3.23. Обмеження часу сесії

Також впроваджено захист від CSRF (Cross-Site Request Forgery). Це так звана «підробка міжсайтових запитів», коли зловмисник змушує користувача виконати небажані дії на сайті, на якому той вже авторизований.

У нас це реалізовано за допомогою Flask-WTF, який автоматично генерує CSRF-токени для захисту форм. У нас цей захист використовується в маршрутах, які обробляють запити (рис. 3.9). Або у файлі скриптів, де CSRF-токен, який додається до кожного POST-запиту, генерується сервером і передається у фронтенд через мета-тег (рис. 3.19). А для глобального CSRF-захисту у файл __init__.py

додається генерція токенів у Flask (рис. 3.23).

```
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect(app)
```

Рисунок 3.23. Захист CSRF

Шифрування даних.

Для шифрування токенів доступу access_token використовується бібліотека cryptography (рис. 3.24), бо вона легка у використанні та підтримує як симетричне, так і асиметричне шифрування.

```
from cryptography.fernet import Fernet # type: ignore

key = Fernet.generate_key()
cipher_suite = Fernet(key)
encrypted_token = cipher_suite.encrypt(user.access_token.encode()
decrypted_token = cipher_suite.decrypt(encrypted_token).decode()
```

Рисунок 3.24. Підключення бібліотеки cryptography

Таким же самим чином шифруються й інші дані проєкту: дані користувачів (email, username тощо) та історія чату.

А при завантаженні нашого додатка на хостинг, можна буде під’єднати SSL-сертифікат, що надає безпечну передачі даних через HTTPS. Зробити це можна буде через Nginx.

Захист бази даних.

Від SQL-ін’єкцій вберігають готові інструменти SQLAlchemy та PostgreSQL (рис 3.25). Здійсненні через них запити автоматично екранують небезпечні символи.

```
user = User.query.filter_by(email=email).first()
```

Рисунок 3.25. Автоматичне екранування даних

Також для забезпечення максимальної безпеки бази даних, можна реалізувати розмежування прав доступу. Цей підхід базується на принципі мінімальних привілеїв, тобто кожен користувач повинен мати доступ лише до тих ресурсів, які

необхідні для виконання його функцій. Це вбереже, наприклад, від несанкціонованих змін чи видалення даних.

Логи і моніторинг

Усі дії користувачів, такі як вхід, вихід, очищення чату, помилки, неуспішні способи входу тощо або інші підозрілі дії, введення та отримання симптомів тощо, логуються (рис. 3.26).

```
# Логування спроби входу
app.logger.info(f"User {username} (ID: {user.id}) logged in successfully.")

# Логування успішного виходу
app.logger.info(f"User {username} (ID: {user_id}) logged out.")

# Логування спроби очищення чату
app.logger.info(f"Chat history cleared for user {current_user.id}.")

# Логування помилки під час очищення чату
app.logger.error(f"Error clearing chat for user {current_user.id}: {e}")

# Логування спроби виходу без авторизації
app.logger.warning("Attempted logout without an authenticated session.")

# Логування неуспішних спроб входу
if failed_attempts[email] > 3:
    app.logger.warning(f"Multiple failed login attempts detected for email {ema
```

А для своєчасного моніторингу активності можна налаштувати email-сповіщення про аномалії. Налаштування таких сповіщень показано на рисунку 3.27.

Захист від атак.

Для захисту від XSS (Cross-Site Scripting), вразливості, яка дозволяє

```
from flask_mail import Mail, Message

mail = Mail(app)
msg = Message("Виявлено підозрізу активність",
              sender="електронна пошта адміністратора",
              recipients=["електронна пошта адміністратора"
msg.body = "Багато невдалих спроб входу."
mail.send(msg)
```

Рисунок 3.27. Сповіщення про аномалії

зловмисниками впроваджувати шкідливі скрипти у погано захищені сайти, які потім виконуються у браузерах інших користувачів, можна використати кілька підходів.

Екранування даних, тобто замінення на знешкодження шкідливих символів та небезпечних HTML-кодів. Такий спосіб використовується, наприклад, у функції обробки даних diagnose, за допомогою бібліотеки bleach (рис. 3.9). Також для цієї мети можна використовувати функцію escape, що робить те саме, що й bleach.

Екранування в шаблонах. Flask автоматично екранує будь-які змінні. Наприклад, якщо в даному шаблоні {{ message }} message містить якийсь скрипт, то браузер відобразить лише текст, а не виконає скрипт.

Захист через HTTP-заголовки. Наприклад, у файл __init__.py є заголовок Content Security Policy (CSP), який обмежує джерела виконуваного коду. Це дозволяє завантаження ресурсів тільки з нашого хосту та забороняє виконання скриптів з інших джерел. Та заголовок X-XSS-Protection, що вказує браузеру блокувати підозрілий контент (рис 3.28).

Рисунок 3.28. Заголовок X-XSS-Protection

Також можна перевіряти довжину надісланого повідомлення і якщо воно буде надто довгим, то вилетить помилка.

А для захисту від DdoS-атак використовується Flask-Limiter, що обмежує частоту запитів (рис. 3.29).

Для вберігання даних від раптових небезпек також було налаштоване **резервне копіювання даних**. У нас використовується інструмент pg_dump – інструмент командного рядка PostgreSQL для створення дампів баз даних. Він дозволяє зберегти копію нашої бази у вигляді SQL-скриптів, або юінарних файлів, які можна використовувати для відновлення.

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(app, key_func=get_remote_address)

@app.route('/diagnose')
@limiter.limit("10 per minute")
def diagnose():
    return jsonify({'message': 'Запит оброблено'})
```

Рисунок 3.29. Використання Flask-Limiter

Вручну резервне копіювання можна зробити за допомогою цієї команди:

pg_dump -U user -d SympDB -F c > /backup/SympDB_$(date +\%F).sql

А для автоматизації усього процесу можна використати cron. Це спеціальна утиліта для операційних систем Unix та Linux, яка дозволяє користувачам виконувати команди або скрипти автоматично в заданий час. Для цього потрібно в редакторі cron, що відкривається командою `crontab -e`, додати команду для автоматичного резервного копііюввання:

0 2 * * * pg_dump -U user -d SympDB -F c > /backup/SympDB_$(date +\%F).sql

При завжди запущеному сервері, або коли сайт буде завантажено на хостинґ, резервне копіювання буде проводитись кожного дня о 2:00 ранку.

Також можна додати стиснення резервних копій, за допомогою gzip, видалення старий копій, а ще можна робити резервні копії конфігураційних файлів бази даних

А для **оновлення програмного забезпечення** у нас є файл залежностей requirements.txt (рис. 3.30), який періодично треба оновлювати за допомогою — `upgrade`.

Оновлення програмного забезпечення дуже важливе, бо від цього залежить робота усього проєкту, тож потрібно оновлювати усі складові.

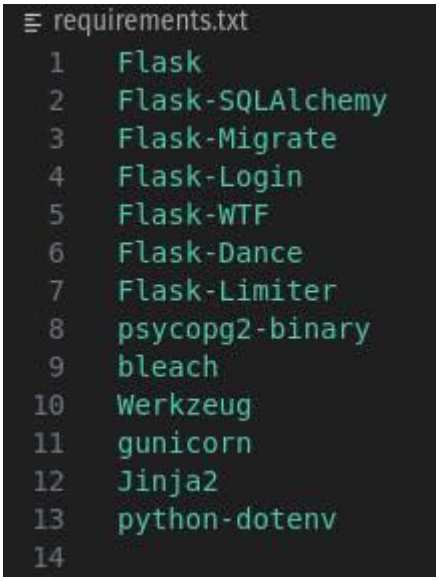

Рисунок 3.30. Залежності проєкту

Висновки до розділу 3

У третьому розділі було реалізовано практичну частину проєкту, спрямовану на створення симптоматичного діагностичного додатка з інтегрованими кіберзахисними механізмами. Усі поставлені завдання, включаючи розробку алгоритму діагностики, реалізацію інтерфейсу користувача, створення функціоналу чату та системи авторизації, було успішно виконано. Особливу увагу приділено впровадженню захисних заходів, таких як шифрування даних, захист від CSRF та обмеження доступу до бази даних, що суттєво підвищило безпеку системи.

Значення отриманих результатів полягає у створенні багатофункціонального та безпечного вебдодатка, здатного не лише забезпечувати точну діагностику за симптомами, але й захищати конфіденційність даних користувачів. Успішне тестування функціоналу підтвердило відповідність системи поставленим вимогам і високий рівень її надійності.

У подальшому вдосконалення цього проєкту може включати розширення алгоритмів діагностики, інтеграцію додаткових захисних механізмів, а також оптимізацію для роботи з великими обсягами даних у реальному часі. Це дозволить створити ще більш ефективну й безпечну систему для користувачів.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було досягнуто поставленої мети – розроблено інтелектуальну систему діагностики на основі симптомів, яка поєднує високий рівень функціональності з інтегрованими механізмами кібербезпеки. Завдяки реалізації проєкту вдалося вирішити всі поставлені завдання, включаючи аналіз проблеми, вибір відповідних технологій, розробку багаторівневої архітектури, створення алгоритму діагностики, інтеграцію користувацького інтерфейсу та впровадження комплексних засобів захисту даних.

Отримані результати мають вагоме практичне значення. Система, розроблена в межах цього дослідження, забезпечує швидкий аналіз введених симптомів із можливістю надання точних попередніх діагнозів, що підвищує доступність медичної інформації для широкого кола користувачів. Високий рівень захищеності персональних даних досягнуто завдяки використанню сучасних технологій, таких як Google OAuth 2.0, шифрування даних, захист від CSRF-атак, а також обмеження частоти запитів для запобігання DDoS-атакам. Зручний, адаптивний та інтерактивний інтерфейс дозволяє користувачам легко взаємодіяти із системою.

Значення проведеного дослідження полягає у створенні багатофункціональної системи, яка не лише покращує процес діагностики, але й демонструє практичну реалізацію ефективних механізмів кіберзахисту в контексті вебзастосунків. Висновки роботи можуть бути використані як основа для розробки подібних систем у різних сферах діяльності.

Перспективи подальших досліджень включають розширення функціональності системи шляхом впровадження більш складних алгоритмів діагностики з урахуванням додаткових параметрів, таких як демографічні дані, історія хвороб та генетичні фактори. Інтеграція із зовнішніми джерелами, такими як медичні API, дозволить автоматично оновлювати базу даних симптомів та захворювань. Вдосконалення кіберзахисту через впровадження квантово-стійких алгоритмів шифрування, систем виявлення аномальної поведінки користувачів та розширення засобів захисту від багатошарових атак підвищить надійність системи. Адаптація

до роботи з великими обсягами даних у реальному часі забезпечить ефективність використання системи для масових аудиторій або в медичних установах. Додатково, покращення користувацького досвіду через розробку мобільних додатків сприятиме ще більшій доступності системи.

Таким чином, проведене дослідження сприяє розвитку сучасних підходів до створення інтелектуальних діагностичних систем, а також підвищенню рівня безпеки вебзастосунків, що є актуальним у контексті глобалізації та цифровізації суспільства.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The History of the First Computer Virus on Windows, Mac, and Linux. URL: https://www.thepcinsider.com/history-of-computer-virus (date of access: 08.10.2024).

2. Події з історії кібербезпеки за 50 років. URL: https://datami.ua/podiyi-z-istoriyi-kiberbezpeki-za-50-rokiv (дата звернення: 10.10.2024)

3. Май Дзя. Дешифрувати. Київ: Сафран, 2019. 368 с.

4. A Short History of Cryptography. URL: http://all.net/edu/curr/ip/Chap2-1.html (date of access: 14.10.2024)

5. RSA Algorithm in Cryptography. URL: https://www.geeksforgeeks.org/rsa-algorithm-cryptography (date of access: 17.10.2024)

6. A brief history of cryptography: Sending secret messages throughout time. URL: https://www.ibm.com/think/topics/cryptography-history (date of access: 19.10.2024)

7. Технології забезпечення безпеки мережевої інфраструктури

8. A Brief History of Antivirus Software. URL: https://fusioncomputing.ca/history-of-antivirus (date of access: 21.10.2024)

9. What is a firewall. URL: https://www.cisco.com/site/us/en/learn/topics/security/what-is-a-firewall.html (date of access: 21.10.2024)

10. CRYSTALS. Cryptographic Suite for Algebraic Lattices. URL: https://pq-crystals.org/kyber/ (date of access: 24.10.2024)

11. Zero-day vulnerabilities. URL: https://www.zero-day.cz (date of access: 27.10.2024)

12. GDPR або General Data Protection Regulation. URL: https://legalitgroup.com/gdpr-novi-eu-tendentsii/ (date of access: 30.10.2024)

13. Stuxnet: перша у світі кіберзброя. URL: https://futurenow.com.ua/stuxnet-persha-u-sviti-kiberzbroya/ (date of access: 02.11.2024)

14. What is hacktivism? URL: https://www.techtarget.com/searchsecurity/definition/hacktivism (date of access:

01.11.2024)

15. Діордіца І. В. Класифікація кіберзагроз та їх легітимація у нормативно-правових актах України. Підприємництво, господарство і право. 2017. № 10. С. 206–211. URL: http://www.pgp-journal.kiev.ua/archive/2017/10/43.pdf (дата звернення: 01.11.2024)

16. Cyber Threats Classifications and Countermeasures in Banking and Financial Sector / A. A. Darem et al. IEEE Access. 2023. P. 1. URL: https://doi.org/10.1109/access.2023.3327016 (date of access: 02.11.2024).

17. Knapp E. D., Langill J. T. Industrial Cyber Security History and Trends. Industrial Network Security. 2015. P. 41–57. URL: https://doi.org/10.1016/b978-0-12-420114-9.00003-4 (date of access: 02.11.2024).

18. Методи та засоби захисту інформації / А. Аль-Амморі, та ін. Системи управління, навігації та зв'язку. Збірник наукових праць. 2024. Т. 1, № 75. С. 38–44. URL: https://doi.org/10.26906/sunz.2024.1.038 (дата звернення: 04.11.2024).

19. Корнієнко Б. Я., Юдін О. К., Снігур О. С. Безпека аутентифікації у web-ресурсах. Ukrainian Information Security Research Journal. 2012. Т. 14, № 1 (54). URL: https://doi.org/10.18372/2410-7840.14.2056 (дата звернення: 04.11.2024).

20. Корченко О. Г., Сіденко В. П., Дрейс Ю. О. Прикладна криптологія: системи шифрування : підручник. Київ: ДУТ, 2014. 448 с.

21. What is EDR vs. XDR? URL: https://www.paloaltonetworks.com/cyberpedia/what-is-edr-vs-xdr (date of access: 10.11.2024)

22. Корченко О. Г., Логінов І. В., Скворцов С. О. Stationary systems of cyberattacks detection and prevention for cyberprotection and cybercounterintelli-gence (by example USA). Ukrainian Scientific Journal of Information Security. 2019. Vol. 25, no. 1. URL: https://doi.org/10.18372/2225-5036.25.13664 (дата звернення: 12.11.2024).

23. Маттес Е. Пришвидшений курс Python / пер. з англ. Київ: Видавнича група КМ-БУКС, 2021. 544 с.

24. Flask Documentation. URL: https://flask.palletsprojects.com/en/stable (date of

access: 13.11.2024)

25. SQLAlchemy Documentation. URL: https://docs.sqlalchemy.org/en/20 (date of access: 13.11.2024)

26. PostgreSQL Documentation. URL: https://www.postgresql.org/docs (date of access: 13.11.2024)

27. Web Documentation. URL: https://developer.mozilla.org/en-US/docs/Web/Tutorials (date of access: 13.11.2024)

28. jQuery API Documentation. URL: https://api.jquery.com (date of access: 15.11.2024)

29. Flask-WTF Documentation. URL: https://flask-wtf.readthedocs.io/en/1.2.x/ (date of access: 15.11.2024)

30. Flask-Limiter Documentation. URL: https://flask-limiter.readthedocs.io/en/stable/ (date of access: 15.11.2024)

31. Google Oauth 2.0 Guide. URL: https://developers.google.com/identity/protocols/oauth2 (date of access: 20.11.2024)

32. Flask-Dance Documentation. URL: https://flask-dance.readthedocs.io/en/latest (date of access: 20.11.2024)