Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки та захисту інформації
_____ Іван ПАРХОМЕНКО
«__» _____ 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

освітня програма *Кібербезпека*
(назва освітньо-професійної програми)

на тему: «Метод виявлення та запобігання шкідливим ботам у Java веб-сервісах»

Виконавець: студент ІІ курсу, групи КБм-21

_____ **Богдан СИДОРЕНКО**
(підпис) (ім'я, прізвище)

	Ім'я, прізвище	**Підпис**
Керівник	Лариса МИРУТЕНКО	
Нормоконтроль	Юрій БАБЕНКО	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«25» жовтня 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача КБм-21 (група) Сидоренка Богдана Миколайовича (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Метод виявлення та запобігання шкідливим ботам у Java веб-сервісах

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Процес забезпечення захисту Java веб-сервісів у контексті бот-атак.
Предмет досліджень	Метод виявлення та запобігання автоматизованій шкідливій активності у веб-додатках.
Мета	Розробка методу для виявлення та запобігання шкідливим ботам у веб-сервісах, створених з використанням Java.

Вихідні дані для проведення роботи	Механізми виявлення та блокування шкідливих ботів у Java веб-сервісах.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна	Розробка методу для виявлення та запобігання ботів, поєднуючи статичні, поведінкові та евристичні механізми захисту.
Практична цінність	Покращення захисту Java веб-сервісів від шкідливої бот-активності.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	25.10.2024 – 29.12.2024
Аналіз літературних джерел	30.12.2024 – 11.02.2025
Ознайомлення з сучасними підходами до безпеки веб-сервісів у Java	12.02.2025 – 20.02.2025
Розгляд нормативно-правових документів щодо захисту інформації у веб-середовищі	21.02.2025 – 24.02.2025
Дослідження загроз, вразливостей і типових бот-атак на веб-додатки	25.02.2025 – 05.03.2025
Аналіз рекомендацій щодо захисту від ботів	06.03.2025 – 09.03.2025
Дослідження існуючих інструментів виявлення та запобігання ботів	10.03.2025 – 18.03.2025
Розгляд рішень Java-екосистеми	19.03.2025 – 22.03.2025
Реалізація власного методу виявлення ботів у Java веб-сервісі	23.03.2025 – 18.04.2025
Тестування працездатності реалізованого методу захисту та аналіз результатів	19.04.2025 – 24.04.2025
Оформлення пояснювальної записки згідно методичних рекомендацій	25.04.2025 – 15.05.2025
Подача пакету документів на розгляд ЕК	15.05.2025 – 19.05.2025

Завдання видала _____ Лариса МИРУТЕНКО
(підпис) (Ім’я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Богдан СИДОРЕНКО
(підпис) (Ім’я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.

Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Метод виявлення та запобігання шкідливим ботам у Java веб-сервісах»: 83 сторінок, 36 рисунків та 2 таблиць, 40 літературних джерел.

Об’єкт дослідження – процес забезпечення захисту Java веб-сервісів у контексті бот-атак.

Мета роботи – розробка методу для виявлення та запобігання шкідливим ботам у веб-сервісах, створених з використанням Java.

Методи дослідження – аналіз HTTP-запитів, поведінкове моделювання користувачів, підходи виявлення бот-активності, CAPTCHA, алгоритми фільтрації трафіку, механізми обробки заголовків запитів у Java веб-сервісах.

У роботі досліджено сучасні загрози, пов’язані з автоматизованою шкідливою активністю у веб-середовищі. Проведено аналіз методів виявлення ботів на основі заголовків HTTP-запитів, частоти звернень, поведінкових аномалій та взаємодії з інтерфейсом. Запропоновано комбінований метод, що дозволяє виявляти ботів у Java веб-сервісах та запобігати атакам на ранніх етапах.

Наукова новизна: розроблено комбінований метод для виявлення та запобігання ботів, поєднуючи статичні, поведінкові та евристичні механізми захисту.

Актуальність теми: Автоматизовані бот-атаки становлять серйозну загрозу безпеці сучасних веб-сервісів. Вони призводять до витоку даних, перевантаження систем і втрати фінансових ресурсів. Стандартні засоби захисту часто не справляються з новими типами загроз. Запропонований метод дозволяє підвищити стійкість Java веб-сервісів до таких атак, поєднуючи кілька механізмів аналізу та фільтрації на різних рівнях взаємодії з клієнтом.

Ключові слова: кібербезпека, шкідливі боти, веб-сервіси, Java, HTTP-фільрація, поведінковий аналіз.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

REST	–	Representational State Transfer
SOAP	–	Simple Object Access Protocol
API	–	Application Programming Interface
(D)DoS	–	(Distributed) Denial-of-Service
HTTP	–	HyperText Transfer Protocol
HTTPS	–	HyperText Transfer Protocol Secure
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
MITM	–	Man-In-The-Middle
CAPTCHA	–	Completely Automated Public Turing test to tell Computers and Humans Apart
CDN	–	Content Delivery Network
ASN	–	Autonomous System Number
DOM	–	Document Object Model
SQL	–	Structured Query Language
ML	–	Machine Learning
TTL	–	Time To Live
WAF	–	Web Application Firewall
JSON	–	JavaScript Object Notation
CORS	–	Cross-Origin Resource Sharing
ATO	–	Account Takeover
IP	–	Internet Protocol
LMS	–	Learning Management System

ЗМІСТ

ВСТУП

Актуальність даної роботи зумовлена необхідністю розробки методу виявлення та запобігання шкідливим ботам, які становлять реальну загрозу для безпеки сучасних веб-сервісів.

Шкідлива бот-активність впливає як на продуктивність систем, так і на конфіденційність даних користувачів. Запропоноване рішення має універсальний характер і може бути впроваджене як у персональних, так і корпоративних веб-системах, що розширює сферу його застосування.

Основною задачею цієї кваліфікаційної роботи є дослідити сучасні механізми виявлення ботів, проаналізувати найбільш ефективні способи їх нейтралізації та запропонувати комплексний підхід, орієнтований на інтеграцію з Java веб-сервісами. Особливу увагу приділено аналізу HTTP-запитів, поведінкових ознак та механізмів CAPTCHA.

Науковою новизною цієї роботи є розробка адаптивного методу, який поєднує існуючі техніки детекції ботів з удосконаленими механізмами обробки запитів, що дозволяє виявляти автоматизовані атаки. Реалізація виконана з використанням Java, що дозволяє легко інтегрувати метод до існуючих веб-додатків.

Майже кожен користувач інтернету стикався з ситуацією, коли сайти стають недоступними, з’являються обмеження на функціонал або виникають складнощі з реєстрацією – усе це може бути наслідками бот-атак. З точки зору користувача – це незручності, а з боку сервісу – це потенційні витрати, втрата даних і репутаційні ризики. Насправді ж боротьба з ботами – складний процес, який вимагає надійних і ефективних механізмів захисту.

Об’єктом дослідження є система безпеки веб-сервісів у контексті захисту від шкідливої бот-активності.

Предметом дослідження є механізми виявлення ботів на основі аналізу HTTP-запитів, заголовків, частоти звернень та поведінкових патернів у Java веб-середовищі.

Стрімкий розвиток цифрових технологій, зокрема хмарних сервісів, електронної комерції та онлайн-платформ, відкриває нові можливості, але й створює нові виклики. Відсутність надійних механізмів протидії ботам може призвести до витоку даних, зниження продуктивності, шахрайства та інших серйозних загроз.

У сучасному світі дедалі більше сервісів автоматизуються – замовлення, підписки, запити до API – і саме тому так важливо відрізнити реального користувача від автоматизованого скрипта. Надмірна активність ботів може викривити аналітику, зірвати рекламні кампанії або стати основою для складніших атак.

Метою даної кваліфікаційної роботи є опис, реалізація та перевірка ефективності методу виявлення та запобігання шкідливим ботам, який ґрунтується на комплексному аналізі запитів і поведінки користувача. Метод має бути легко адаптованим до Java веб-додатків та здатним працювати в умовах реального навантаження.

Більшість пристроїв, які використовуються сьогодні – смартфони, ноутбуки, телевізори, планшети – мають доступ до Інтернету, а отже й до веб-сервісів. Забезпечення їхньої стабільності та безпеки є ключовим завданням. Проте досі багато хто вважає, що захист – це складно, дорого або взагалі неефективно. Такий підхід лише підсилює ризики.

Україна, як і більшість країн, перебуває під постійною загрозою кіберінцидентів. Значна частина атак має автоматизований характер – боти сканують сайти, шукають вразливості, здійснюють спам-реєстрації чи DDoS-атаки.

Результати роботи, що включають як теоретичні підходи, так і реалізований захист у Java, можуть бути використані розробниками веб-сервісів для підвищення рівня безпеки своїх систем. Запропонований метод може стати додатковим рівнем захисту в архітектурі інформаційної безпеки.

РОЗДІЛ 1
АНАЛІЗ ЗАГРОЗ ВІД ШКІДЛИВИХ БОТІВ

1.1 Роль веб-сервісів у цифровому середовищі

У сучасному цифровому світі, де інформація є стратегічним ресурсом, веб-сервіси стали основним механізмом для забезпечення ефективної, масштабованої та стандартизованої взаємодії між програмними системами, організаціями, пристроями й людьми. Їх роль у цифровому середовищі важко переоцінити: від обслуговування найпростіших запитів користувача до побудови складних інфраструктур розподілених обчислень, веб-сервіси пронизують практично всі рівні цифрової взаємодії. Веб-сервіси, по суті, стали мовою, якою «спілкуються» різні програмні компоненти між собою у цифровому світі. Ця «мова» є універсальною, стандартизованою та незалежною від конкретної реалізації чи технології, що дозволяє об’єднувати найрізноманітніші системи – від мобільних додатків і вебпорталів до хмарних платформ і вбудованих пристроїв. Завдяки цьому, веб-сервіси виконують роль сполучної тканини цифрового простору, забезпечуючи безперервність, взаємодію, доступність функцій і даних для будь-якої точки цифрової екосистеми. Вони усувають бар’єри між платформами, дозволяють стандартизовано обмінюватися повідомленнями, трансформують фрагментовані інформаційні системи у єдине інтегроване середовище, де кожен компонент виконує свою функцію, залишаючись при цьому незалежним.

Історія розвитку веб-сервісів розпочалася у 1990-х роках із появи концепції сервіс-орієнтованої архітектури. Тоді виникла необхідність у створенні механізмів обміну даними між ізольованими програмами, що функціонували в межах однієї організації або навіть між організаціями. Відповіддю на ці потреби стало створення протоколу SOAP і мови опису інтерфейсів WSDL. Ці технології забезпечували структурований, формальний підхід до побудови систем інтеграції, який дозволяв чітко описувати структуру запитів, типи переданих даних, послідовність викликів та

очікувані відповіді. Завдяки суворій схемі XML і опису WSDL, системи могли заздалегідь знати, як саме взаємодіяти одна з одною, що забезпечувало високу сумісність та передбачуваність у роботі. Такий підхід був особливо ефективним у великих корпоративних середовищах, де інтеграція між різними модулями (наприклад, фінансовим, логістичним, управління персоналом) вимагала надійності, стабільності та відповідності нормативним вимогам. Проте зі зростанням кількості клієнтських пристроїв і необхідністю гнучкої масштабованості ці обмеження стали заважати подальшому розвитку.

Подальший розвиток Інтернету та мобільних додатків виявив обмеження SOAP-підходу, зокрема надмірну складність, громіздкість XML і неефективність при інтеграції з фронтенд-додатками. Це стимулювало перехід до REST – архітектурного стилю, який базується на принципах HTTP і дозволяє створювати прості, ефективні, масштабовані API. REST API стали універсальним стандартом інтеграції: їх використовують як великі хмарні платформи (AWS, Google Cloud, Azure), так і мобільні додатки, веб-застосунки, IoT-пристрої, системи електронної комерції, фінансові сервіси та багато інших типів програмного забезпечення. REST дозволяє створювати легко доступні й стандартизовані точки доступу до функціоналу систем, що суттєво спрощує як розробку клієнтів, так і супровід бекендів. Гнучкість, простота реалізації, підтримка кешування, масштабованість і природне використання HTTP-запитів зробили REST одним із найуспішніших рішень для створення розподілених систем, що взаємодіють через Інтернет.

Окрім REST і SOAP, з’явилися альтернативні підходи, що вирішують специфічні задачі. Наприклад, gRPC – це високопродуктивний фреймворк для взаємодії між мікросервісами, який використовує Protocol Buffers замість JSON для серіалізації даних і підтримує двостороннє стрімінгове спілкування через HTTP/2. GraphQL, розроблений Facebook, дозволяє клієнтам формувати гнучкі запити лише до потрібних полів, що особливо цінно для мобільних застосунків з обмеженим трафіком. WebSockets, у свою чергу дають змогу встановити постійне двостороннє з’єднання між клієнтом і сервером, що є надзвичайно важливим для систем, які потребують передачі даних у реальному часі. Наприклад, біржові платформи,

онлайн-чати, ігрові сервіси, служби відстеження геопозиції, push-нотифікації – усі ці рішення використовують WebSockets для забезпечення мінімальних затримок, миттєвого оновлення інформації та асинхронної взаємодії. На відміну від класичних HTTP-запитів, які ініціюються клієнтом, WebSocket-з'єднання дозволяє серверу самостійно надсилати дані, щойно з'являється нова інформація. Це значно знижує навантаження на систему та покращує користувацький досвід у динамічних застосунках.

З технологічної точки зору веб-сервіси можна класифікувати на декілька типів: публічні (доступні всім розробникам), приватні (використовуються всередині організації), партнерські (відкриті лише для певного кола інтеграторів). Це розмежування має велике значення для безпеки, контролю доступу, ліцензування, логування та відповідності регуляторним вимогам. Наприклад, фінансові установи відкривають API виключно для сертифікованих партнерів відповідно до вимог PSD2 та Open Banking Standard.

Сучасні веб-сервіси стали ядром цифрової трансформації підприємств. Вони забезпечують інтеграцію між CRM, ERP, системами аналітики, платформами звітності, мобільними додатками. Наприклад, при оформленні онлайн-замовлення в інтернет-магазині активуються десятки API-запитів: перевірка доступності товару, обрахунок доставки, підтвердження платежу, оновлення залишків на складі, формування рахунку, надсилання push-сповіщення тощо. Кожен із цих запитів є окремим веб-сервісом, який функціонує незалежно та масштабовано, дозволяючи оновлювати, розгортати або масштабувати його без впливу на інші частини системи. Такий підхід забезпечує високу гнучкість в управлінні архітектурою, спрощує технічне обслуговування та сприяє підвищенню стійкості всієї системи до збоїв. У результаті бізнес отримує можливість швидко адаптуватися до змін ринку, впроваджувати нові функції й інтегрувати сторонні сервіси без потреби в суттєвих переробках основного програмного забезпечення.

У банківській сфері веб-сервіси забезпечують зв'язок між фронт- і бекофісом, між мобільними додатками й основними банківськими системами, між партнерами та ядром банку. Наприклад, мобільний банкінг – це сукупність сотень API, що

відповідають за перевірку балансу, переказ коштів, верифікацію користувача, отримання виписок, обробку карткових операцій, push-сповіщення, а також обмін інформацією з державними системами (наприклад, податковими службами або сервісами верифікації особи).

У сфері охорони здоров'я веб-сервіси дозволяють поєднувати медичні інформаційні системи, електронні медичні картки, аптеки, лабораторії та державні реєстри. Через API відбувається обмін результатами аналізів, направленнями, електронними рецептами, інформацією про вакцинацію.

В освітній сфері веб-сервіси стали каталізатором впровадження дистанційного навчання. Навчальні платформи, електронні щоденники, системи тестування, онлайн-курси взаємодіють через API, що дозволяє синхронізувати розклад, оцінки, контент, відеозаписи, індивідуальні кабінети, системи контролю відвідуваності та успішності. Наприклад, LMS типу Moodle, Canvas, Google Classroom надають API для підключення зовнішніх модулів: систем прокторингу, аналітики навчання, генерації звітів.

У державному управлінні роль веб-сервісів особливо важлива. Електронні державні послуги, системи реєстрів, кабінети громадян, міжвідомча взаємодія – усе це реалізується за допомогою API. Наприклад, у системі «Дія» в Україні десятки державних сервісів поєднані в єдиний цифровий простір, де API забезпечують запит та обмін інформацією між Мін'юстом, Державною податковою службою, Пенсійним фондом, ЦНАПами, банками та іншими структурами. Це дозволяє скоротити час обробки звернень, підвищити прозорість, забезпечити інтегровану цифрову взаємодію між державою та громадянами. Така централізована архітектура на основі веб-сервісів дає змогу не лише оптимізувати роботу окремих установ, а й побудувати єдину цифрову екосистему, в якій обмін даними відбувається без участі людини, а користувач отримує комплексну послугу «в один клік». У результаті значно зменшується кількість помилок, викликаних людським фактором, скорочується бюрократія, підвищується довіра громадян до цифрових послуг. Зокрема, під час пандемії, а згодом і в умовах повномасштабної війни в Україні, ефективне функціонування таких сервісів як «Дія» стало критично важливим для отримання

фінансової допомоги, довідок, подання заяв, комунікації з державними органами – і все це відбувалося на основі веб-сервісної архітектури, що забезпечує надійність, масштабованість і швидкість.

У контексті архітектури програмного забезпечення веб-сервіси стали основою побудови сучасних гнучких систем, що масштабуються. Найпоширенішою архітектурною моделлю останніх років є мікросервісна архітектура. У ній кожен сервіс є самостійним компонентом, який реалізує одну бізнес-функцію, має власну базу даних і може розгортатися незалежно від інших. Веб-сервіси виступають основним засобом зв’язку між такими компонентами. Наприклад, при реалізації онлайн-магазину окремі мікросервіси можуть відповідати за каталог товарів, управління кошиком, обробку платежів, реєстрацію користувачів, управління відгуками чи логістику. Кожен із цих мікросервісів взаємодіє з іншими виключно через стандартизовані веб-сервіси, що дозволяє змінювати або оновлювати одну частину системи без впливу на інші. Такий підхід забезпечує високу модульність, спрощує тестування, пришвидшує розгортання нових функцій і дає змогу масштабувати лише ті компоненти, які відчувають найбільше навантаження. Наприклад, під час розпродажів можна збільшити ресурси лише для сервісу, що обробляє замовлення, не зачіпаючи систему рекомендацій або профілі користувачів.

Мікросервісна архітектура надає численні переваги: масштабованість, гнучкість, fault tolerance, незалежне оновлення компонентів. Водночас вона вимагає високої зрілості інфраструктури, зокрема засобів моніторингу, логування, трасування, оркестрації. Веб-сервіси в такій архітектурі повинні бути високонадійними, швидкими і безпечними. Надійна аутентифікація, авторизація на рівні запитів, rate limiting мають бути реалізовані за найкращими практиками, аби гарантувати безперебійну роботу навіть у випадках часткових відмов або пікового навантаження. Надійність кожного компонента досягається через дублювання критичних вузлів, автоматичне перемикання на резервні сервіси, а також контроль часу відповіді, що дозволяє оперативно ідентифікувати та усунути проблеми.

Ще одним сучасним трендом стала serverless-архітектура, де замість розгортання постійних серверів розробники створюють окремі функції, що

виконуються лише за потреби (наприклад, AWS Lambda, Google Cloud Functions). Такі функції часто виступають як веб-сервіси, які обробляють події, взаємодіють із чергами повідомлень, базами даних, сторонніми API. Serverless підхід дозволяє значно знизити витрати на інфраструктуру, але вимагає ретельного підходу до моніторингу продуктивності, cold start latency, обмежень на використання ресурсів та максимального часу виконання. Крім того, у serverless-архітектурі важливо забезпечити належну обробку помилок, повторні виклики при збої, а також підтримку ідентитету та безпеки, оскільки кожна функція виконується ізольовано. Незважаючи на свої переваги, такі як швидке масштабування, serverless вимагає високого рівня дисципліни у проєктуванні логіки та взаємодії між компонентами, що часто призводить до комбінування підходів у гібридних архітектурах.

Окремо слід згадати event-driven архітектуру, де веб-сервіси взаємодіють через події. Такий підхід широко використовується у фінансових біржах, e-commerce платформах, інтернеті речей. Наприклад, у системі логістики подія «створення замовлення» може запустити ланцюжок дій: перевірку наявності товару, резервування, виставлення рахунку, виклик служби доставки – кожен з етапів реалізується окремим сервісом, що підписаний на певний тип подій у брокері повідомлень. Веб-сервіси, що реалізовані в межах подієво-орієнтованої архітектури, повинні бути здатними швидко реагувати на зміну стану системи та взаємодіяти з іншими сервісами в режимі реального часу. Їхня ефективність залежить від здатності правильно обробляти події – як успадковані, так і новостворені – та забезпечувати гарантовану доставку повідомлень. Для цього використовуються механізми підтвердження отримання, обробки повторів, черги з пріоритетами. Надійність таких веб-сервісів є ключовою, оскільки навіть один пропущений тригер може порушити всю бізнес-логіку процесу. Тому вони тісно інтегруються з системами обробки повідомлень, сховищами подій, а також аналітичними сервісами, що дозволяють виявляти «вузькі місця» та оптимізувати ланцюги подій.

Веб-сервіси активно впливають і на цифрову економіку в цілому. З’явився термін «API Economy» – концепція, згідно з якою API є не просто технічним інтерфейсом, а повноцінним економічним активом. Компанії відкривають свої API,

формують платні підписки, створюють партнерські програми, контролюють SLA, вбудовують метрики успішності. Це дає змогу масштабувати бізнес без значного залучення людських ресурсів. Приклади – Stripe, Twilio, Plaid, RapidAPI – компанії, чия бізнес-модель повністю ґрунтується на API.

Іншим значущим трендом є інтеграція веб-сервісів з технологіями штучного інтелекту. Наприклад, сервіси розпізнавання облич, мови, перекладу тексту, генерації відповідей, аналізу тональності працюють як API, які можна підключити до свого застосунку. Веб-сервіси стали каналом доставки інтелектуальних можливостей.

Крім того, зростає значення веб-сервісів у контексті Internet of Things. Пристрої – від домашніх сенсорів до індустріальних датчиків – взаємодіють через API з хмарними платформами, аналітичними системами, мобільними додатками. Веб-сервіси дозволяють обробляти дані в реальному часі, реагувати на події, приймати автоматизовані рішення. Така архітектура широко використовується в smart home, агротехнологіях, логістиці, промисловому виробництві, системах енергоменеджменту.

Зрештою, роль веб-сервісів не обмежується лише технічною реалізацією. Це інструмент побудови цифрових екосистем, цифрової держави, інноваційного бізнесу. Вони лежать в основі таких понять як Smart City, e-Governance, e-Health, цифровий суверенітет. Їх розвиток визначає конкурентоспроможність країни, здатність до цифрової інтеграції в глобальний ринок, рівень послуг, що надаються громадянам і бізнесу.

Таким чином, веб-сервіси у цифровому середовищі виступають не лише технічним інструментом, а й основою для формування нових економічних, соціальних і культурних моделей. Вони забезпечують гнучкість, відкритість, доступність і масштабованість цифрової інфраструктури, сприяючи побудові стійких, інноваційних і людиноцентричних цифрових систем.

Для глибшого розуміння ролі веб-сервісів у цифровому середовищі доцільно провести аналітичне порівняння найпоширеніших підходів до побудови API. REST, GraphQL і gRPC є домінантами сучасного технологічного стеку, але кожен має свої

сильні сторони та сфери застосування. REST, як класичний і найпоширеніший підхід, простий у реалізації, має широку підтримку інструментів і легко масштабується. Його недоліки проявляються при роботі з надлишковими обсягами даних – клієнт отримує повну відповідь навіть тоді, коли йому потрібна лише частина інформації. Це призводить до перевитрати трафіку, збільшення часу відповіді та навантаження на канал зв'язку – особливо критично в умовах мобільних мереж або для клієнтів з обмеженими ресурсами. Саме ці недоліки стали поштовхом до розвитку альтернатив, таких як GraphQL, який дозволяє оптимізувати передачу даних відповідно до реальних потреб клієнта.

GraphQL – це запитна мова, яка дозволяє клієнту самостійно формувати структуру відповіді, зменшуючи надлишковість і кількість запитів. Такий підхід ідеально підходить для мобільних додатків і складних клієнтських інтерфейсів. Однак реалізація GraphQL вимагає значно складнішої логіки на бекенді, а також потребує додаткової уваги до контролю доступу, кешування та безпеки.

gRPC є сучасною реалізацією RPC від Google, що забезпечує мінімальну затримку, двосторонній стрімінг і високу продуктивність. Він активно застосовується у високонавантажених мікросервісних системах, наприклад, у телекомунікаціях, геймдеві, фінансовому аналізі. Недоліком gRPC є обмежена підтримка браузерами без проксі-рішень, складність дебагінгу й потреба у спеціальних клієнтах для роботи з Protocol Buffers.

З погляду майбутнього, веб-сервіси стають важливим елементом Web3 – децентралізованої архітектури Інтернету. Наприклад, для взаємодії з блокчейн-мережами використовуються такі сервіси, як Infura, Alchemy, які надають API для зчитування стану блокчейну, надсилання транзакцій, підпису даних. Це відкриває можливості для побудови децентралізованих застосунків, які можуть функціонувати незалежно від традиційних бекендів.

Edge Computing – ще один напрям, де веб-сервіси використовуються для обробки даних ближче до джерела. У таких системах веб-сервіси розгортаються не в хмарі, а на фізичних пристроях – шлюзах, промислових контролерах, мобільних вузлах. Це дозволяє знизити затримки, зменшити обсяг передаваних даних,

підвищити автономність. Типові сценарії: автономне керування транспортом, машинне зір, швидке реагування на аварійні події.

Таким чином, веб-сервіси є ключовим інструментом побудови цифрової цивілізації. Вони об’єднують системи, людей, пристрої, алгоритми. Від забезпечення доступу до банківського рахунку – до керування енергосистемою мегаполіса; від цифрового кабінету громадянина – до штучного інтелекту в смартфоні. Усі ці сценарії базуються на веб-сервісах – стандартизованих, безпечних, масштабованих та відкритих. Їхня роль лише зростатиме, формуючи нову парадигму взаємодії у цифровому світі.

1.2 Види шкідливих ботів і типові загрози

У сучасному цифровому середовищі використання шкідливих ботів стало системною загрозою для веб-сервісів різного масштабу – від невеликих сайтів до великих хмарних платформ. Шкідливі боти – це програми, які автоматизовано виконують дії, що зазвичай вимагають участі людини. Якщо деякі боти відіграють позитивну роль (наприклад, сканери пошукових систем або моніторингові агенти), то велика частина ботів працює на шкоду: викрадає дані, порушує доступність ресурсів, шахрайськи взаємодіє з функціоналом сайту, або використовує веб-сервіс як інструмент подальших атак.

Історично перші шкідливі боти були відносно примітивними – це були прості скрипти, які надсилали HTTP-запити до форм входу або реєстрації без емулювання браузера. Такі запити легко відрізнялись від звичайної активності через відсутність cookies, JavaScript, а також підозрілі або відсутні User-Agent заголовки. Однак із розвитком антибот-засобів зловмисники почали створювати дедалі складніші рішення. Сучасні шкідливі боти нерідко побудовані на базі headless-браузерів, здатних виконувати JavaScript, взаємодіяти з DOM, кліками, формами, імітувати поведінку користувача.

Більше того, існує цілий підпільний ринок інструментів для автоматизації атак. Сервіси типу «Bots-as-a-Service» або панелі керування бот-мережами дозволяють

навіть нефахівцям запускати складні атаки без жодного програмування. Часто такі інструменти продаються на форумах даркнета у вигляді щомісячної підписки. Наприклад, існують сервіси для масового створення облікових записів, скрейпінгу контенту, розсилки спаму. Це свідчить про те, що кіберзлочинність набуває рис платформи – з інтерфейсами, клієнтською підтримкою, тестовим періодом, навіть системами лояльності.

У контексті атак шкідливі боти відіграють ключову роль як інструмент автоматизації. Зокрема, одним із найпоширеніших сценаріїв є credential stuffing – коли бот перебирає тисячі комбінацій логінів і паролів, використовуючи бази даних з попередніх витоків. Успішне потрапляння до облікового запису дає змогу зловмиснику викрасти кошти, отримати доступ до приватних даних або використати акаунт як інструмент атаки. Credential stuffing відрізняється від звичайного brute force тим, що не перебирає паролі випадково, а працює з реальними обліковими даними. Це робить атаку ефективнішою, і водночас складнішою для виявлення, адже запити схожі на легітимні спроби входу.

Ще одним прикладом є scraping боти, які автоматично збирають контент із сайтів: ціни, зображення, описи, відгуки, контактні дані. Для деяких сайтів – особливо маркетплейсів, туристичних агрегаторів, або сайтів нерухомості – таке скрейпінг є критичною загрозою. Він не лише порушує авторські права, але й може призвести до втрати конкурентної переваги, зниження трафіку, порушення унікальності контенту. Сучасні скрейпери можуть обходити захисти через проксі-мережі, зберігати cookie, змінювати браузерні параметри для обходу fingerprinting-захисту.

Особливе місце займають spam боти. Вони використовуються для масової публікації небажаного контенту: в коментарях, формах зворотного зв’язку, форумах, навіть у чатах підтримки. Основною метою є або посилення пошукових позицій через спам-посилання, або реклама шахрайських сайтів, або зараження користувачів через фішингові посилання. Деякі спам-боти навіть навчаються автоматично обходити базові перевірки типу «обов’язкове поле» або «людська валідація».

Цікаво, що inventory hoarding боти або «reserving bots» масово використовуються у сфері e-commerce. Їх завдання – зарезервувати товар або квитки на подію в кошику, що робить їх тимчасово недоступними для звичайних покупців. Такі боти часто діють у зв’язці зі scalping-ботами, які вже здійснюють покупку після релізу – наприклад, у випадках релізів популярних кросівок, концертних квитків або нових моделей PlayStation. Бізнес втрачає лояльність користувачів, справжні клієнти не можуть купити продукт, а перекупники продають його в декілька разів дорожче.

Окрему загрозу становлять боти для захоплення акаунтів – так звані Account Takeover. Вони спрямовані на отримання повного контролю над обліковим записом жертви. На відміну від простого credential stuffing, ATO-боти можуть комбінувати кілька методів: викрадення токенів автентифікації, кукі-файлів, використання MITM-атак, підміни реферера або ін'єкції сесій. Особливо складно виявити ATO-ботів у випадках, коли вони діють через headless-браузери, імітуючи поведінку користувача у всіх деталях – аж до рухів миші, швидкості введення паролів та взаємодії з елементами DOM.

Після захоплення акаунта можливі різні сценарії використання: крадіжка грошей, зміна адреси доставки, витік конфіденційної інформації, шантаж або використання облікового запису як точки входу в корпоративну мережу. Через високу вартість облікових записів на чорному ринку (особливо у сферах фінансів, страхування, ритейлу) зловмисники часто створюють ботнети, що паралельно тестують тисячі скомпрометованих облікових даних на десятках платформ.

Ще один різновид ботів – Click fraud боти, які штучно генерують кліки на рекламні банери. Метою може бути злив рекламного бюджету конкурентів, підвищення власного доходу (наприклад, через партнерські програми), або псування аналітики маркетингових кампаній. Click fraud часто використовується у поєднанні з IP-спуфінгом, геолокаційною підміною та поведінковими шаблонами, які імітують зацікавленість користувача. Збитки компаній від такого виду атак сягають мільярдів доларів щорічно. За даними аналітиків Juniper Research, до 30% глобального рекламного трафіку може бути фальсифіковано ботами.

У фінансовому секторі небезпеку становлять carding боти, які автоматизовано підбирають комбінації номерів банківських карток, термінів дії та CVV-кодів. Ці боти тестують тисячі комбінацій на сайтах з онлайн-платежами, визначаючи дійсні карти. Навіть при високому рівні захисту картка може бути «протестована» через сервіс з меншою перевіркою, наприклад, сервіс пожертв або підписки. Надалі валідні карти використовуються для шахрайських покупок або продаються на даркнет-форумах.

Не менш поширеними є боти для DoS- або DDoS-атак. Вони створюють навантаження на сервер або обмежений API-ендпоінт шляхом надсилання величезної кількості запитів. Це призводить до вичерпання обчислювальних ресурсів, затримок у роботі або повної недоступності сервісу. Окремі боти реалізують «повільні» DoS-атаки, коли сервер утримується в активному стані шляхом часткового, затягнутого запиту. Через те, що такі атаки не створюють пікових навантажень, вони складно виявляються стандартними засобами моніторингу.

У практиці протидії бот-атакам дедалі частіше використовується класифікація ботів за рівнем складності, яка допомагає визначити рівень загрози та відповідні засоби захисту. До найпростішої категорії, першого рівня, належать звичайні скрипти, які здійснюють прямі HTTP-запити без підтримки JavaScript, cookies або взаємодії з DOM. Боти другого рівня вже використовують headless-браузери або автоматизовані браузери, як-от Selenium, і здатні виконувати JavaScript-код, проте їм часто бракує повноцінної емуляції поведінки користувача. Третій рівень складності становлять боти, які здатні імітувати базову поведінку людини: рухи миші, кліки, прокручування сторінки, а також вводити випадкові затримки між діями задля підвищення правдоподібності. Найвищий, четвертий рівень, включає так звані псевдолюдські боти, що використовують методи машинного навчання для адаптації до контексту взаємодії. Такі боти здатні аналізувати структуру сторінки, враховувати логіку інтерфейсу та навіть реагувати на зміни у формі введення.

Розуміння рівнів складності є критично важливим для побудови ефективної системи захисту. Так, для протидії ботам першого і другого рівня зазвичай достатньо

базових механізмів перевірки, таких як валідація заголовків User-Agent, наявність cookies або правильна структура HTTP-запиту. Натомість для виявлення і блокування ботів третього і четвертого рівня необхідне впровадження складніших механізмів: поведінкового аналізу, застосування CAPTCHA, виявлення аномалій у шаблонах взаємодії, обмеження сесій, геолокаційної фільтрації, rate limiting тощо. Саме багаторівнева структура захисту дозволяє підвищити ефективність виявлення сучасних загроз і мінімізувати ризики обходу системи ботами з високим рівнем адаптації. Слід також зазначити, що сучасні шкідливі боти не лише атакують поодинокі ресурси. Часто вони стають інструментами у розподіленій інфраструктурі атак, де один тип бота здійснює сканування, інший – атаку, третій – моніторинг відповіді, четвертий – повторну спробу обходу. Таким чином формується бот-екосистема з розподіленою відповідальністю, яка є стійкою до виявлення та блокування. Це ускладнює захист і вимагає від компаній впровадження комплексного підходу до ідентифікації та управління трафіком.

Однією з ключових характеристик сучасного шкідливого ботнету є його архітектура. Залежно від цілей атаки, рівня організації та засобів протидії, бот-мережі можуть мати різну структуру, що визначає їхню стійкість, складність виявлення та ефективність. Найбільш простою формою є централізовані ботнети, в яких усі заражені пристрої підключаються до одного або кількох керівних серверів, від яких отримують команди. Така архітектура є відносно простою у реалізації, проте має серйозну вразливість – у разі виявлення або блокування керуючого сервера вся мережа втрачає контрольованість і стає неефективною.

Для підвищення стійкості до виявлення все частіше використовуються децентралізовані ботнети, в яких кожен пристрій не лише отримує команди, а й передає їх далі іншим вузлам. У таких системах відсутній єдиний центр, і керування мережею відбувається у форматі peer-to-peer, що значно ускладнює її локалізацію та нейтралізацію. Еволюційним розвитком цієї моделі є гібридні ботнети, які поєднують централізоване керування із децентралізованими елементами. Такі мережі здатні включати в себе різні типи пристроїв – від десктопів до IoT-модулів і мобільних телефонів. Для забезпечення стійкості вони можуть мати резервні

C2-сервери, використовувати інфраструктуру CDN для маскування або динамічно змінювати свої мережеві адреси.

Окремий тип становлять ботнети, побудовані на базі хмарних платформ. У такому випадку зловмисники використовують ресурси легальних хмарних сервісів, таких як Amazon Web Services, Google Cloud або Microsoft Azure, отримані через скомпрометовані або неправомірно створені акаунти. Завдяки цьому атаки маскуються під легітимний трафік з високим рівнем довіри, що значно ускладнює їх виявлення та блокування. Крім того, такі ботнети легко масштабуються, динамічні та здатні швидко перебудовуватися.

Незалежно від типу архітектури, сучасні ботнети часто мають розвинену інфраструктуру самооновлення. Це означає, що окремі вузли можуть завантажувати нові версії програмного забезпечення, змінювати конфігурацію, адаптувати поведінкові шаблони залежно від реакції захисних механізмів. У результаті протидія таким системам потребує не лише технічних засобів, а й стратегічного підходу, оскільки боротьба з ними більше нагадує протистояння з адаптивним противником.

Значна частина шкідливих ботів бере участь у складених атаках, виступаючи як перший етап проникнення або автоматизована ланка в ланцюжку дій:

У фішингових кампаніях боти можуть створювати або поширювати фейкові сторінки входу, автоматично збирати введені користувачами логіни, паролі, OTP-коди.

Для обходу двофакторної автентифікації використовуються боти-перехоплювачі, які в режимі реального часу чекають, поки користувач уведе код, і швидко використовують його для входу.

В APT-ланцюжках боти можуть сканувати порти, виявляти версії сервісів, і навіть надсилати експлойти автоматично, поки інша частина інфраструктури контролює отримані результати.

У фазі збору розвідданих боти маскуються під легітимні користувацькі агенти, щоб пройти обхід правил WAF та отримати доступ до прихованих API, sitemap, внутрішніх панелей.

На рівні атак «через довіру» боти використовуються для «підфарбовування» рейтингів, залишення фальшивих відгуків, автоматизації соціального інжинірингу через чат або соцмережі.

Боти, таким чином, не обов’язково самостійно реалізують весь вектор атаки, але виступають як інструменти підготовки, виконання або маскування. Їхня здатність діяти швидко, повторювано, непомітно й масово робить їх універсальними у руках зловмисника.

Шкідливі боти становлять одну з найскладніших і найдинамічніших загроз для сучасних веб-сервісів. Їхній розвиток від простих скриптів до високорівневих, машинно-керованих агентів із можливістю імітації людської поведінки свідчить про технологічну еволюцію атак. Завдяки розподіленій архітектурі, гнучкості налаштувань, здатності маскуватися під легітимний трафік та обходити навіть складні захисні механізми, боти перетворились із маргінального інструмента в повноцінний елемент кіберзлочинної екосистеми.

Їх використання виходить далеко за межі простого зловмисного скрейпінгу чи атак на авторизацію: сучасні боти є учасниками фішингових кампаній, складених атак APT, економічного саботажу, маніпуляції аналітикою та конкурентної розвідки. При цьому здатність таких агентів динамічно адаптуватись до умов середовища, використовувати методи машинного навчання та обходити традиційні перевірки на «людяність» ускладнює їх виявлення.

1.3 Вплив бот-атак на функціонування веб-сервісів

Бот-атаки є одним із найбільш недооцінених, але вкрай впливових чинників, що порушують стабільність, продуктивність та безпеку веб-сервісів. Незважаючи на свою, на перший погляд, неагресивну форму (зазвичай це прості HTTP-запити), шкідливі боти здатні викликати значні технологічні, економічні та репутаційні наслідки для будь-якого онлайн-ресурсу. В умовах цифрової економіки, де користувацький досвід, доступність і надійність відіграють ключову роль, навіть

незначне втручання ботів може призвести до втрати конкурентних переваг, скорочення доходів або правових санкцій.

Один із найпомітніших наслідків – зниження продуктивності веб-сервісу. Високочастотна активність ботів створює додаткове навантаження на серверну інфраструктуру. Це проявляється у збільшенні часу відповіді, сповільненні завантаження сторінок, нестабільності API. Особливо небезпечним це стає в пікові періоди (сезонні розпродажі, реєстраційні кампанії, квиткові запуски), коли справжні користувачі не можуть отримати доступ до ресурсу через перевантаження, створене ботами. У деяких випадках бот-активність навіть не має на меті DoS, але в результаті викликає схожий ефект.

Другою важливою проблемою є спотворення бізнес-аналітики. Веб-сервіси широко використовують системи аналітики для збору даних про поведінку користувачів: сторінки, що переглядаються, кліки, конверсії, джерела трафіку, географія відвідувачів тощо. Присутність ботів у трафіку знижує точність цих даних, що призводить до неправильних управлінських рішень, неефективного таргетингу, неадекватного планування маркетингових кампаній. Наприклад, боти можуть штучно збільшити кількість переглядів сторінки, створити враження високого інтересу до певного товару або, навпаки, викликати необґрунтовану тривогу про відтік користувачів.

Особливу небезпеку становлять економічні збитки, спричинені поведінкою ботів у комерційних сценаріях. У сфері електронної торгівлі це може бути блокування запасів товарів у кошиках, масова реєстрація фейкових акаунтів для отримання знижок, викуп дефіцитної продукції через scalping-ботів. У рекламному бізнесі – click fraud, що призводить до некоректного списання бюджетів. У фінансовому секторі – масові атаки на форми входу, що створюють додаткове навантаження на інфраструктуру та сприяють втраті клієнтів через зниження рівня довіри.

Ще один впливовий аспект – використання веб-сервісу як вхідної точки в більшу інформаційну систему. Боти можуть тестувати слабкі місця в логіці обробки запитів, ідентифікувати некоректно захищені API, взаємодіяти з публічними

інтерфейсами, які дають доступ до внутрішніх ресурсів. Таким чином, навіть на перший погляд незначний бот може стати першою ланкою в APT-атаці або атаці на ланцюг постачання. Такі сценарії особливо небезпечні для мікросервісної архітектури, де одна вразливість може порушити роботу кількох критичних компонентів системи.

Не менш важливою є репутаційна шкода, яку може завдати масова бот-активність. Наприклад, веб-сайт, що регулярно недоступний або працює повільно через ботів, втрачає довіру клієнтів. Якщо платформа дозволяє залишати відгуки, боти можуть використовувати її для генерації фальшивих негативних рецензій, штучного пониження рейтингу або політично мотивованого тиску. У соціальних мережах боти часто використовуються для масових репостів, накрутки голосувань або спотворення інформаційного середовища. Результатом є не лише втрата аудиторії, а й зниження позицій у пошукових системах, які також реагують на підозрілу активність.

Окремо слід згадати правові та нормативні наслідки. Бот-атаки, що призводять до витоку персональних даних або компрометації облікових записів, можуть стати підставою для штрафів згідно з GDPR, CCPA або національним законодавством. У разі, якщо компанія не впровадила достатні заходи захисту, вона може бути визнана відповідальною за збитки, завдані третім сторонам. Окрім того, навіть без прямого витоку даних, організація може втратити право на сертифікацію через високий рівень підозрілих сесій і незахищеність інтерфейсів.

Таким чином, вплив шкідливих ботів на веб-сервіси є багатовимірним: від зниження доступності та продуктивності до підриву довіри, втрати даних і прямих збитків. Особливо небезпечним є той факт, що більшість таких впливів реалізуються повільно, поступово – і не завжди помітні у стандартному технічному моніторингу. Це вимагає від організацій не лише технологічної підготовленості, а й усвідомлення ризиків на рівні бізнесу, менеджменту та стратегії розвитку.

Висновки за розділом 1

Проведене у першому розділі дослідження дозволило сформувати системне уявлення про природу веб-сервісів як ключового інструменту сучасної цифрової взаємодії, а також дати всебічну характеристику шкідливим ботам – одному з найактуальніших типів загроз для інформаційної безпеки веб-інфраструктури.

Було встановлено, що веб-сервіси на сьогодні є основою інтеграції цифрових систем у різних сферах: від електронної комерції, фінансів, охорони здоров'я та освіти – до державного управління. Завдяки відкритій архітектурі, стандартизованим протоколам (REST, GraphQL, gRPC), високій масштабованості та здатності до взаємодії, веб-сервіси забезпечують основу для цифрової трансформації як окремих компаній, так і цілих держав. Однак саме ці характеристики, що роблять веб-сервіси ефективними, водночас створюють значні виклики в галузі кібербезпеки – зокрема, через їхню вразливість до автоматизованих бот-атак.

Аналіз типів шкідливих ботів дозволив класифікувати основні загрози, зокрема такі як credential stuffing, scraping, spam, DoS, scalping, click fraud, carding, ATO, fake registration та inventory hoarding. Було з'ясовано, що сучасні шкідливі боти здатні адаптуватися до контексту, маскувати свою активність, використовувати headless-браузери, proxy-мережі, антифінгерпринтинг, а також елементи машинного навчання. Їх функціонування часто здійснюється через розгалужені бот-мережі з децентралізованою або гібридною архітектурою, що ускладнює їх виявлення й блокування.

Окрему увагу було приділено комплексному впливу бот-активності на функціонування веб-сервісів. Було встановлено, що негативні наслідки проявляються не лише в технічному вимірі (перевантаження інфраструктури, погіршення продуктивності, вразливість до DoS-атаки), а й в економічному (збитки від фальшивих транзакцій, накрутки кліків, витрат на інфраструктуру), аналітичному (спотворення даних, на яких базуються бізнес-рішення), юридичному (порушення регламентів захисту персональних даних), а також репутаційному (втрата довіри з боку користувачів та партнерів).

Загальна тенденція, яка простежується в описаних кейсах, – перехід від примітивних ботів до складних, квазіінтелектуальних агентів, що здатні не просто

виконувати скрипти, а й приймати рішення в реальному часі, обходити захист, вести складні багатокрокові взаємодії з веб-додатками. Це обумовлює необхідність багаторівневої, гнучкої, адаптивної та постійно оновлюваної стратегії виявлення й протидії бот-активності.

Таким чином, проведений аналіз підтвердив гіпотезу про високий рівень загроз, що виникають у результаті дії шкідливих ботів, та обґрунтував доцільність і актуальність подальшого дослідження в напрямі створення надійних, ефективних і технологічно гнучких методів виявлення та запобігання бот-атакам.

РОЗДІЛ 2
РОЗГЛЯД МЕХАНІЗМІВ ДЛЯ ВИЯВЛЕННЯ ТА ПРОТИДІЇ ШКІДЛИВИМ БОТАМ

2.1 Огляд методів аналізу HTTP запитів і мережевих ознак

Аналіз HTTP-запитів є одним із базових і водночас найефективніших підходів до виявлення шкідливої активності на рівні веб-сервісу. Оскільки саме через HTTP протокол реалізується основна взаємодія між клієнтом та сервером, будь-яке відхилення в структурі або поведінці запитів може бути маркером бот-атаки. Веб-сервери отримують тисячі або навіть мільйони запитів щодня, і не всі з них надсилаються реальними користувачами. Частина запитів – це автоматизовані звернення ботів, які прагнуть отримати доступ до ресурсу, здійснити несанкціоновану активність або протестувати вразливості. У таких умовах аналіз метаданих, які супроводжують запити, стає ключовим інструментом кіберзахисту.

Першим і найважливішим джерелом інформації є заголовки HTTP-запиту. Найбільш інформативним серед них виступає User-Agent. Цей заголовок повідомляє серверу, з якого клієнта – браузера або іншого ПЗ – було здійснено звернення. У випадку з легітимними браузерами User-Agent виглядає як довгий рядок, що включає назву браузера, його версію, тип операційної системи, іноді розширення або модулі. Наприклад, типовий браузер Chrome передає складний, багаторівневий User-Agent. Водночас шкідливі боти часто або зовсім не передають цей заголовок, або підставляють найпростіші шаблони на кшталт curl/7.64.1, python-requests або Java/1.8.0, що чітко вказує на автоматизований характер запиту. Іноді зловмисники навмисне підставляють відомі браузерні User-Agent, проте відсутність відповідності між User-Agent і рештою поведінкових параметрів все одно дозволяє виявити обман.

Крім User-Agent, важливим є заголовок Referer, який інформує про джерело переходу. У справжніх користувачів він автоматично формується браузером і містить адресу сторінки, з якої було здійснено перехід. У ботів часто цей заголовок відсутній

або неприродно однаковий для всіх запитів. Так само Origin – заголовок, що вказує на домен, з якого надійшов запит, – є обов’язковим у браузерному середовищі при запитах до API або при роботі з CORS. Його відсутність у POST-запиті є вагомим індикатором неінтерактивного середовища.

Окремо слід згадати й про інші, менш помітні заголовки, які все ж несуть у собі важливу інформацію: Accept, Accept-Encoding, Accept-Language, DNT, Sec-CH-UA та інші. Їх присутність, структура та черговість можуть бути проаналізовані на предмет відповідності типовому набору браузера. Браузери мають характерний набір заголовків, який важко точно імітувати вручну. Отже, навіть якщо заголовки присутні, але сформовані в нестандартній послідовності або з помилками, це викликає підозру. Аналіз заголовків – це не просто перевірка наявності конкретного поля, а порівняння шаблону всього HTTP-запиту з типовими профілями, сформованими для різних типів клієнтів.

Крім заголовків, можна виявити аномалії і на рівні мережевого середовища. Наприклад, IP-адреса, з якої надходить запит, може належати до хмарного хостингу або відомого пулу проксі-серверів. У більшості випадків реальні користувачі мають динамічні або регіонально обмежені IP, пов’язані з локальними провайдерами. Натомість бот-мережі часто працюють з дата-центрів, де IP-адреси мають характерний відбиток ASN. Аналізуючи ASN або геолокацію IP-адреси, можна зробити висновок про її ймовірне використання в бот-мережі. Якщо запити з подібного IP надходять із надмірною частотою або одночасно з багатьох підмереж, можна запідозрити наявність скоординованої атаки.

Ще однією важливою ознакою є аналіз частоти запитів. Шкідливі боти зазвичай працюють без затримок і надсилають десятки або сотні запитів за хвилину. Це дозволяє їм за короткий проміжок часу перевірити сотні варіантів логіну, атакувати форму коментарів або спробувати обійти систему CAPTCHA. Визначення IP-адрес із надмірною кількістю запитів за одиницю часу дозволяє оперативно виявити та ізолювати потенційну загрозу.

Також характерною поведінковою рисою ботів є спроба доступу до стандартних URL-адрес, таких як /admin, /wp-login.php, /phpmyadmin, /config, які

найчастіше не відображаються у звичайному функціоналі користувача. Виявлення повторюваних звернень до цих маршрутів, особливо з різних IP, є сигналом про роботу автоматизованого сканера вразливостей.

Таким чином, аналіз HTTP-запитів у поєднанні з дослідженням мережевих ознак дає змогу сформувати ефективний фільтр для виявлення підозрілої активності ще до того, як бот зможе взаємодіяти з бізнес-логікою веб-додатку. Це є основою для побудови систем раннього реагування, адаптивного обмеження доступу, а також реалізації механізмів CAPTCHA, honeypot або повного блокування IP на рівні сервера.

2.2 Досідження поведінковий та евристичний аналіз користувача

У сучасному цифровому середовищі, де значна частина трафіку веб-сервісів формується не людьми, а автоматизованими агентами – скриптами, ботами, парсерами, тестовими середовищами та шкідливими програмами – питання відокремлення реальних користувачів від ботів стає першочерговим. Звичайні підходи виявлення ботів, такі як перевірка HTTP-заголовків або IP-обмеження, часто виявляються недостатніми, адже новітні боти можуть імітувати людську поведінку на дуже високому рівні. Саме тому особливого значення набуває поведінковий аналіз користувача, який базується на фіксації і інтерпретації динаміки взаємодії з веб-ресурсом. На відміну від сигнатурних або блокуючих методів, поведінковий підхід враховує реальний контекст дій користувача: як він рухає мишкою, як заповнює поля, які події генерує у браузері, як довго перебуває на сторінці, наскільки випадковою є його поведінка тощо.

Згідно з численними дослідженнями, навіть найпростіші дії реальної людини залишають після себе характерний цифровий «відбиток». Наприклад, рухи миші не є рівномірними – вони містять інерцію, дрібні коливання, коригування траєкторії. Натискання клавіш відрізняються випадковими затримками, іноді з помилками, іноді з паузами для читання тексту. Людина нерідко змінює фокус вікна, перемикається між вкладками, скролить вгору-вниз у пошуках інформації. Навпаки, бот зазвичай

демонструє передбачувану, швидку та системну поведінку: надсилає форму миттєво після рендерингу сторінки, не генерує жодної події миші або клавіатури, не взаємодіє з DOM-елементами, виконує ті самі дії у точному порядку та з однаковими часовими проміжками. Саме ці закономірності й дозволяють системам захисту формувати евристики – тобто емпіричні правила, що сигналізують про аномальну або автоматизовану активність.

Однією з найбільш очевидних ознак ботоподібної поведінки є використання headless-браузерів – середовищ, які запускають браузерні рушії без графічного інтерфейсу. Такі інструменти, як Puppeteer, Selenium або Playwright у headless-режимі, дають змогу обходити частину фронтенд-захисту, проте все одно залишають сліди: відсутність подій focus/blur, специфічна поведінка властивостей navigator.webdriver, незвичні значення об'єктів window та document. Навіть якщо бот намагається маскуватися під реального користувача, його неможливо змусити повністю відтворити всі властивості реального сеансу, і це відкриває простір для виявлення.

Ще одним прикладом застосування евристичного підходу є механізм honeypot – приховані поля у формах, які не видно справжньому користувачеві, але які можуть бути заповнені ботом через незнання або скриптову обробку всіх інпутів. Ці поля мають стилі display: none, visibility: hidden, або виключені з логіки фокусування. Якщо такі поля виявляються заповненими – це явна ознака автоматизованої взаємодії. Аналогічним методом є виявлення невідповідностей у послідовності подій: наприклад, коли певна дія виконується до початку взаємодії з елементами інтерфейсу, без будь-якого руху миші або клавіатурних подій. Такі ознаки нехарактерні для людини та можуть використовуватись як ознака ботоподібної активності. Ці евристики є легкими у впровадженні, не впливають на зручність користування для справжніх відвідувачів і забезпечують базовий рівень захисту ще до глибшого аналізу.

Поведінковий аналіз також включає фіксацію часових інтервалів між діями користувача. Реальна людина витрачає певний час на ознайомлення з контентом, навігацію, читання, переміщення курсора, введення інформації та прийняття рішень.

На противагу цьому, автоматизовані системи можуть виконувати дії вкрай швидко й синхронізовано. Надто короткі проміжки між подіями – особливо тими, що зазвичай потребують когнітивного зусилля – можуть сигналізувати про скриптову активність. Тому аналіз темпу й послідовності дій дає змогу виявити неприродну поведінку. Додатково, спостереження за подіями JavaScript – наприклад, onkeydown, onmousemove, onfocus, onscroll – дозволяє оцінити, чи мала місце реальна взаємодія з інтерфейсом. Відсутність таких подій у контексті активності часто є непрямим, але надійним сигналом автоматизації.

Окрему увагу заслуговує побудова повноцінної моделі поведінки користувача протягом сесії. Тут враховується кількість і тип подій, частота кліків, довжина активної взаємодії, тривалість між діями, шаблони пересування по сайту та навіть візерунки руху миші. Наприклад, якщо користувач кожного разу натискає на одну й ту саму кнопку з однаковим інтервалом або переміщується по сторінках у надмірно впорядкованому порядку – це може свідчити про автоматизацію. Реальна людська поведінка має хаотичніші риси: одні читають повільно, інші швидко скролять або залишають сторінку відкритою без активності. Такі моделі дозволяють формувати індивідуальні оцінки ризику для кожного користувача – так званий behavioral score – що використовується для адаптивного реагування на потенційні загрози.

У деяких випадках системи також інтегрують машинне навчання для автоматичного класифікування сесій як «людських» або «ботоподібних». Такі моделі тренуються на великих наборах логів і навчаються розрізняти закономірності, які складно описати вручну. Наприклад, нейронні мережі можуть враховувати понад сотню різних ознак, включаючи кількість подій, тип пристрою, швидкість взаємодії, структуру DOM, використання скриптів тощо. У складних випадках застосовуються гібридні підходи, де спочатку працюють евристичні правила, потім – поведінкові сигнали, і вже після цього – ML-класифікатор. Це дозволяє балансувати між продуктивністю, точністю та гнучкістю.

Нарешті, важливо підкреслити, що поведінковий та евристичний аналіз не завжди застосовується для негайного блокування користувача. Часто вони служать першим етапом адаптивного захисту: якщо активність виглядає підозрілою,

користувачу може бути запропоновано пройти CAPTCHA, підтвердити email або виконати інші дії, які важко автоматизувати. Такий підхід дозволяє не втрачати реальних користувачів через хибні спрацьовування, водночас створюючи серйозні перепони для ботів. Поведінковий аналіз у цьому випадку є не просто інструментом фільтрації, а динамічним механізмом адаптації системи до умов ризику – що робить його надзвичайно цінним у сучасному захисті веб-сервісів.

2.3 Аналіз адаптивних механізмів захисту

У боротьбі зі шкідливими ботами одним із ключових напрямів розвитку є не лише фіксовані правила фільтрації чи блокування, а й застосування адаптивних методів реагування, які враховують контекст поведінки користувача, його IP-репутацію, час активності, а також інші чинники. Такий підхід забезпечує гнучкішу та точнішу протидію автоматизованим загрозам і дозволяє зменшити кількість помилкових спрацювань.

Одним із базових елементів адаптивного захисту є обмеження частоти запитів. Цей механізм передбачає встановлення меж на кількість дозволених запитів від одного клієнта за певний проміжок часу. Наприклад, для неавторизованих користувачів може бути встановлений ліміт у 10 запитів на хвилину, тоді як для авторизованих або платних клієнтів – значно вищий. Такий підхід дозволяє ефективно обмежувати ботів, які намагаються здійснювати масові запити з метою сканування або brute-force атак.

Іншим важливим інструментом є CAPTCHA – тест, що дозволяє розрізняти людей і автоматизовані системи. У сучасних умовах найбільш поширеними є Google reCAPTCHA v2 та v3. Вони надають можливість як явно просити користувача вирішити візуальне або інтерактивне завдання (наприклад, вибрати всі зображення з пішохідними переходами), так і здійснювати невидиму перевірку на основі взаємодії користувача зі сторінкою (scrolling, mouse movement, час затримки перед кліком тощо). Інтеграція CAPTCHA дозволяє додати точку верифікації перед критичними

діями – наприклад, реєстрацією, логіном або надсиланням форм – і тим самим знизити ризик зловживань.

Проте статичне впровадження CAPTCHA може негативно впливати на користувацький досвід. Саме тому все більше популярності набуває так званий пом’якшений захист, який передбачає застосування CAPTCHA лише для підозрілих IP-адрес або в разі виявлення аномальної поведінки. Наприклад, якщо користувач перевищив ліміт запитів або занадто швидко заповнив форму, замість негайного блокування його запиту сервіс може перенаправити його на сторінку з CAPTCHA. Якщо перевірка буде пройдена успішно – користувача буде пропущено далі, якщо ні – IP-адресу буде додано до чорного списку. Це дозволяє зберегти баланс між безпекою та зручністю використання системи.

Ще одним прикладом адаптивного механізму є динамічне оновлення порогів. Замість жорстко закодованих значень, система може самостійно адаптуватися до зміни навантаження – наприклад, в години пік послаблювати обмеження або, навпаки, посилювати їх під час виявлення підозрілої активності. Для цього можуть використовуватися прості евристики або навіть алгоритми машинного навчання, які оцінюють поведінкові патерни в реальному часі.

Ефективність адаптивного захисту значно зростає за рахунок централізованого логування і моніторингу. Збір даних про підозрілі запити, спрацювання CAPTCHA, частоту блокувань і інші індикатори дозволяє аналітикам виявляти нові шаблони атак і оперативно реагувати на них. Крім того, такі дані можна повторно використовувати для побудови whitelist/blacklist механізмів або коригування параметрів захисту.

Таким чином, адаптивний захист є ефективним поєднанням кількох стратегій – від базових фільтрів до динамічних перевірок, що дозволяє веб-сервісам протистояти сучасним загрозам і одночасно забезпечити комфортний користувацький досвід. Його застосування є обов’язковим етапом у побудові безпечної архітектури веб-сервісів у відкритому середовищі Інтернету.

Висновки за розділом 2

У межах другого розділу було здійснено ґрунтовний аналіз сучасних підходів до виявлення та нейтралізації шкідливих ботів на рівні веб-сервісів, зосереджуючись на тих методах, які дозволяють ефективно відрізняти легітимну активність користувачів від автоматизованих дій. У ході дослідження було детально розглянуто декілька напрямів, які є ключовими у контексті розробки комплексної системи захисту: аналіз HTTP-запитів, поведінковий та евристичний аналіз користувачів, а також механізми адаптивного реагування на виявлену підозрілу активність.

Першим важливим кроком є статичний та динамічний аналіз вхідних HTTP-запитів. Було показано, що значна кількість ботів демонструє характерні аномалії у структуруванні та змісті заголовків запиту, зокрема – спрощені або відсутні поля User-Agent, Referer, Origin, а також спроби імітувати заголовки браузера без відповідної послідовності дій на фронтенді. Такі запити часто генеруються з бібліотек на зразок curl, requests, httpclient, які не реплікують реальну браузерну поведінку. Водночас, перевірка таких аспектів, як відповідність User-Agent типу пристрою, часові проміжки між запитами, правильність навігаційного ланцюга переходів, дозволяє вже на цьому рівні фільтрувати значну частину підозрілого трафіку. Встановлення порогових значень, регулярні оновлення списків допустимих або заборонених шаблонів, а також кореляція з даними геолокації чи IP reputation дозволяють автоматизувати цю перевірку в режимі реального часу.

Окрему увагу було приділено поведінковому аналізу користувачів як інструменту другого рівня виявлення ботоподібної активності. Поведінкові сигнали, такі як тривалість перебування на сторінці, наявність взаємодії з елементами DOM, варіативність кліків, природність рухів миші, затримки між подіями – усе це формує унікальний поведінковий профіль, який значно важче підробити, ніж заголовки запиту. Особливо актуальним є виявлення headless-режимів, що широко застосовуються бот-мережами, оскільки вони дають змогу обходити візуальні перевірки, але залишають характерні сліди у середовищі виконання JavaScript, зокрема змінені або відсутні властивості navigator, window, некоректна обробка

подій тощо. Поєднання аналізу таймінгів, шаблонів навігації та реакцій на інтерфейс створює комплексну картину взаємодії, що підвищує точність розпізнавання автоматизованих сесій.

Евристичні механізми, у свою чергу, базуються на накопиченому досвіді та типових ознаках шкідливої активності. Вони дають змогу оперативно реагувати на загрози, які ще не формалізовані у вигляді сигнатур чи моделей. Приклади таких методів – приховані honeypot-поля у формах, виявлення аномально швидкої взаємодії з інтерфейсом, аналіз послідовності подій, що не відповідає логіці дій звичайного користувача. Ці прості й ефективні інструменти особливо добре зарекомендували себе у боротьбі з так званими «загальнодоступними ботами» – інструментами, що використовуються масово та не мають глибокого захисту від виявлення.

Нарешті, особливе місце в архітектурі захисту займають механізми адаптивного реагування, які забезпечують баланс між безпекою та зручністю користування. Система не повинна блокувати користувача на основі одного підозрілого сигналу – натомість вона може змінювати рівень довіри динамічно, залежно від сукупності факторів. Наприклад, при виявленні атипової поведінки можна тимчасово обмежити доступ до частини функцій, вимагати проходження CAPTCHA, повторної автентифікації або підтвердження особистості. Таким чином, система отримує змогу не лише реагувати на вже відомі загрози, а й запобігати потенційним атакам з високим рівнем гнучкості та масштабованості.

Узагальнюючи вищезазначене, можна зробити висновок, що ефективна система виявлення шкідливих ботів не базується на єдиному підході, а потребує комбінації різних механізмів – від базової перевірки HTTP-заголовків до побудови складних моделей поведінкової аналітики. Саме мультифакторний підхід дозволяє досягти високої точності розпізнавання, мінімізувати хибні спрацьовування та забезпечити безперервну еволюцію захисних механізмів.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ МЕТОДУ ВИЯВЛЕННЯ І БЛОКУВАННЯ БОТІВ У JAVA ВЕБ-СЕРВІСІ

3.1 Розробка архітектури системи та вибір технологій

Було реалізовано прототипову систему захисту веб-додатку від шкідливої автоматизованої активності, орієнтованої на виявлення ботів та мінімізацію їхнього впливу. Рішення побудоване як повноцінний веб-додаток на Java з використанням фреймворку Spring Boot, що дозволяє ефективно реалізовувати архітектуру мікросервісного або монолітного типу із чистим поділом відповідальностей.

В основі архітектури лежать принципи чистої архітектури та розділення відповідальностей, що забезпечують логічне відокремлення шарів обробки: HTTP-рівень (фільтрація запитів), бізнес-логіка (перевірки на бот-поведінку), інтеграція із зовнішніми сервісами (Google reCAPTCHA) та рівень доступу до даних (база даних IP-блокувань).

Одним із центральних елементів архітектури є фільтрація HTTP-запитів до того, як вони досягають рівня контролерів. Це дозволяє блокувати потенційно небезпечний трафік ще до активації бізнес-логіки. У системі реалізований фільтр, який перевіряє IP-адресу, аналізує HTTP-заголовки, виконує rate-limiting та інші перевірки. Таке архітектурне рішення значно підвищує рівень безпеки веб-додатку.

Робота системи реалізована у вигляді послідовного проходження HTTP-запиту через набір перевірок, які ілюструє відповідна блок-схема [Додаток А]. Весь потік обробки умовно можна поділити на декілька фаз: початкова обробка, фільтрація, поведінковий аналіз, CAPTCHA-перевірка та реакція (блокування або пропуск).

Усі запити до системи надходять від зовнішніх клієнтів. Це можуть бути як легітимні користувачі, що працюють через браузер, так і автоматизовані скрипти або боти, які надсилають HTTP-запити напряму. На цьому етапі не виконується жодної

обробки, але всі подальші етапи спрямовані на розмежування цих двох типів трафіку.

Запит потрапляє у вбудований веб-сервер Spring Boot, який використовує Apache Tomcat як HTTP-обробник. Цей сервер приймає всі запити, забезпечує маршрутизацію до фільтрів, контролерів, та виконує базові HTTP-операції. На цьому рівні важливо, що запити проходять крізь попередньо налаштовані фільтри безпеки.

Центральним елементом фільтрації є клас RequestFilter, який реалізує інтерфейс OncePerRequestFilter. Цей фільтр гарантує, що запит буде перевірено лише один раз за обробку. Усередині фільтра послідовно виконується кілька важливих перевірок, що допомагають виявити потенційно небезпечну активність.

Першим кроком у фільтрі є перевірка IP-адреси клієнта. Якщо IP вже внесено до чорного списку, запит негайно блокується, і система повертає відповідь про відмову в доступі.

Request Header Validator аналізує обов’язкові заголовки HTTP-запиту: User-Agent, Referer, Origin. Відсутність або аномальні значення вказаних заголовків можуть свідчити про використання скриптів або headless-браузерів. Якщо заголовки не відповідають очікуваним шаблонам, запит маркується як потенційно небезпечний.

Далі система перевіряє частоту запитів з одного IP-джерела. Якщо кількість запитів перевищує встановлений поріг за певний період часу, система визначає таку активність як підозрілу. Реалізація механізму rate limiting відбувається через бібліотеку Caffeine із TTL-кешем, який забезпечує швидкодію та стабільність.

Після проходження базових перевірок активується модуль поведінкового аналізу. Він перевіряє наявність JavaScript-подій, руху миші, кліків, а також взаємодії з елементами сторінки. Відсутність таких сигналів означає, що запит міг бути згенерований ботом, а не реальним користувачем.

Додатковим рівнем є honeypot – приховане поле у формах, яке не бачить справжній користувач. Якщо бот заповнює таке поле, це надійний сигнал, що перед нами автоматизована система. У такому випадку користувач блокується.

Цей механізм порівнює час між завантаженням сторінки та надсиланням форми. Якщо форма надіслана занадто швидко, це вважається неприродною поведінкою. Такі запити вважаються підозрілими і відправляються на CAPTCHA-перевірку.

Кожен компонент системи виконує окрему функцію. Наприклад, сервіс IpBlockService працює з базою даних, зберігаючи та перевіряючи IP-адреси, які слід заблокувати. Цей компонент використовує Spring Data JPA для доступу до реляційної бази даних, що робить реалізацію простою та ефективною.

Для перевірки частоти запитів реалізовано RateLimiterService, що працює з кешуючою структурою Caffeine. Завдяки цьому сервіс не перевантажується запитами з одного джерела, а ботоподібна активність може бути обмежена за допомогою TTL та фіксації кількості спроб у певний період часу.

Якщо деякі попередні перевірки не дали остаточної відповіді або запит визначено як підозрілий, користувач перенаправляється на сторінку з CAPTCHA. Система інтегрує Google reCAPTCHA v2, яка дозволяє розмежувати людей і ботів без додаткових обтяжень.

Якщо CAPTCHA не пройдена або активність користувача визнається бот-типовою, IP-адреса блокується. Сервіс IpBlockService зберігає цю інформацію у базу, включаючи причину, час, тип порушення. Таким чином система поступово формує чорний список і зменшує навантаження від шкідливих джерел у майбутньому.

Для тестування і розгортання системи використано H2 Database, яка дозволяє зручно запускати додаток у середовищі розробника. При переході до продакшн-режиму можна легко інтегрувати PostgreSQL або іншу СУБД завдяки абстракціям Spring Boot.

Усі компоненти проєкту мають високий ступінь ізоляції, що дозволяє їх незалежне тестування. Це дає змогу створити надійне рішення, що забезпечує виявлення ботів без впливу на продуктивність основного функціоналу веб-додатку.

У процесі реалізації архітектури системи було використано низку технологій та бібліотек, які забезпечили гнучкість, масштабованість, зручність підтримки та

безпечну обробку HTTP-запитів. Таке поєднання сучасних засобів дозволило не лише спростити розробку, а й забезпечити високу ефективність і стабільність роботи системи в умовах навантажень, характерних для реального продакшен-середовища. Окрему увагу було приділено вибору інструментів, які добре інтегруються між собою, мають активну спільноту розробників і стабільну підтримку.

Завдяки правильному підбору компонентів стало можливим реалізувати логіку блокування, моніторингу, кешування та перевірки користувачів у рамках єдиної архітектури. При цьому кожен з модулів залишився незалежним, що забезпечує простоту масштабування та внесення змін у майбутньому. Перелік основних використаних технологій наведено в таблиці 3.1.

Таблиця 3.1

Таблиця використаних технологій та бібліотек

Категорія	**Технологія / Бібліотека**	**Призначення**
Фреймворк веб-додатку	Spring Boot 3.4.3	Основний фреймворк для побудови веб-застосунків
Web модуль	spring-boot-starter-web	Робота з HTTP, контролерами, REST API
Безпека	spring-boot-starter-security	Базова безпека HTTP-запитів
Шаблонізація	spring-boot-starter-thymeleaf	Генерація HTML
Підключення до БД / ORM	spring-boot-starter-data-jpa	Робота з базами даних через JPA
Інтерфейс JPA	jakarta.persistence-api	API для JPA, необхідний для компіляції
База даних	h2	Вбудована БД H2 для збереження чорного списку
Міграції БД	flyway-core	Міграція схем БД (версіонування таблиць)
Кешування	caffeine 3.1.8	Бібліотека для високопродуктивного кешу
Логування	SLF4J + Spring Boot Logging	Стандартизоване логування
Анотації / скорочення коду	lombok 1.18.30	Генерація геттерів, конструкторів, логерів та ін.

Основою усього застосунку є фреймворк Spring Boot версії 3.4.3. Його вибір зумовлений здатністю швидко запускати веб-додатки без надлишкової конфігурації, підтримкою автоматичної ініціалізації компонентів та модульністю. Spring Boot

забезпечує інтеграцію з веб-сервером Tomcat, інструментами безпеки, базами даних, а також бібліотеками для логування та тестування.

Для реалізації обробки HTTP-запитів, контролерів та фільтрів використано бібліотеку spring-boot-starter-web. Цей модуль надає можливості для створення REST API, обробки маршрутизації запитів, додавання фільтрів для раннього перехоплення запитів, таких як RequestFilter, та інтеграції з MVC-компонентами.

У частині формування інтерфейсів, зокрема HTML-сторінки з CAPTCHA, використовувався шаблонізатор Thymeleaf, який інтегрується через spring-boot-starter-thymeleaf. Ця бібліотека дозволяє легко формувати динамічні HTML-сторінки без необхідності використовувати JavaScript на серверній стороні, що особливо зручно при генерації форм і відповідей.

Для забезпечення захисту HTTP-ендпоінтів у додатку було використано spring-boot-starter-security. Він забезпечує базову конфігурацію безпеки, фільтрацію запитів, захист від CSRF-атак, підтримку HTTPS та налаштування доступу до ресурсів за допомогою ролей і фільтрів доступу. Навіть у випадках, коли користувача потрібно лише перевірити через CAPTCHA, доступ до деяких ендпоінтів має бути чітко контрольований.

Збереження інформації про заблоковані IP реалізується через JPA-репозиторії, що підключаються через spring-boot-starter-data-jpa. Цей модуль забезпечує зручну роботу з базою даних у вигляді об’єктів і дозволяє виконувати CRUD-операції без написання SQL-запитів. Додатково підключено jakarta.persistence-api – специфікацію, яка формалізує роботу з JPA, щоб забезпечити коректну компіляцію та поведінку всіх об’єктів.

На етапі розробки та тестування для зберігання даних використовувалась вбудована реляційна база даних H2. Вона дозволяє запускати застосунок без додаткового налаштування серверу баз даних, а також забезпечує швидкий перехід до PostgreSQL чи MySQL завдяки абстрагуванню роботи з БД через JPA.

Для керування схемою бази даних застосовується Flyway – бібліотека, яка дозволяє виконувати контрольовані міграції таблиць. У контексті нашої системи

вона використовується для створення й оновлення таблиці, у якій зберігається інформація про заблоковані адреси.

Для кешування частоти запитів та реалізації механізму rate limiting обрано бібліотеку Caffeine. Вона забезпечує високу продуктивність кешу, налаштування часу життя елементів та обмеження обсягу, а також мінімальні затрати пам’яті. Це дозволяє зберігати останні запити від IP та швидко визначати перевищення лімітів.

Для уніфікованого логування у системі використовується SLF4J у поєднанні з Spring Boot Logging. Це забезпечує єдиний підхід до логування, який дозволяє фіксувати винятки, підозрілу активність, виклики фільтрів і реакцію на підозрілі запити.

Також активно використовується бібліотека Lombok, яка значно скорочує обсяг коду, необхідного для написання моделей, сервісів і DTO. Вона автоматично генерує геттери, сеттери, конструктори, методи toString і логери, що позитивно впливає на підтримуваність і швидкість розробки.

У результаті реалізовано архітектуру, що поєднує в собі простоту налаштування, гнучкість, модульність, розширюваність і надійність. Це дозволяє інтегрувати рішення в існуючі сервіси без критичних змін, а також масштабувати систему відповідно до зростання навантаження.

3.2 Впровадження механізм блокування IP-адрес

У межах реалізації системи захисту від ботів ключовим елементом є механізм блокування IP-адрес, які виявлено як джерела підозрілої або шкідливої активності. Реалізація цього механізму складається з кількох компонентів, включно з базою даних, міграційним скриптом Flyway, репозиторієм для доступу до даних, сервісним шаром для бізнес-логіки блокування, а також фільтром, який здійснює перевірку вхідних запитів.

Основу цього механізму становить таблиця бази даних blacklisted_ips, структура якої представлена у вигляді ER-діаграми (рис. 3.1). Вона слугує для

збереження інформації про IP-адреси, що були заблоковані в процесі функціонування системи.

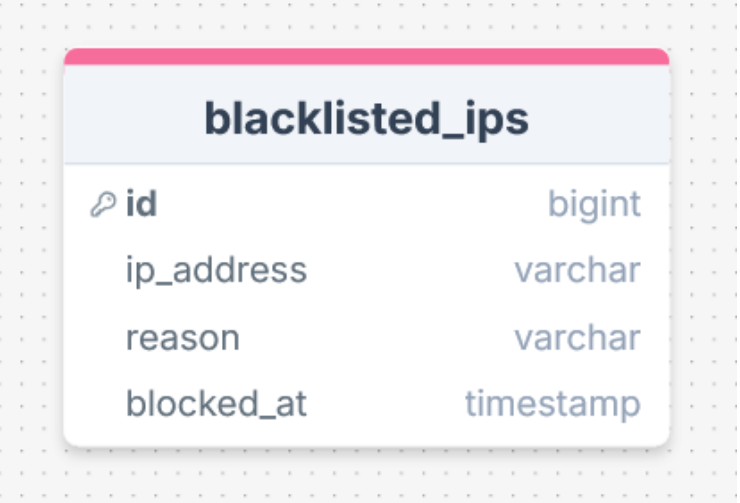


Рисунок 3.1 – ER-діаграма таблиці blacklisted_ips

База даних містить таблицю з назвою blacklisted_ips, яка призначена для зберігання інформації про всі IP-адреси, що були заблоковані в процесі функціонування системи. Таблиця має такі поля: id (унікальний ідентифікатор запису), ip_address (рядок з IP-адресою), blocked_at (мітка часу блокування у форматі timestamp), reason (текстова причина блокування. Структура таблиці дозволяє системі зберігати як постійні, так і тимчасові блокування та виконувати динамічну перевірку їхньої актуальності при кожному запиті.

Створення таблиці відбувається автоматизовано за допомогою Flyway-міграцій. У скрипті (рис. 3.2) створено таблицю blacklisted_ip із вказаними полями, типами даних, первинним ключем по полю id. Використання Flyway гарантує контрольовану міграцію структури БД між середовищами розробки та продакшену.

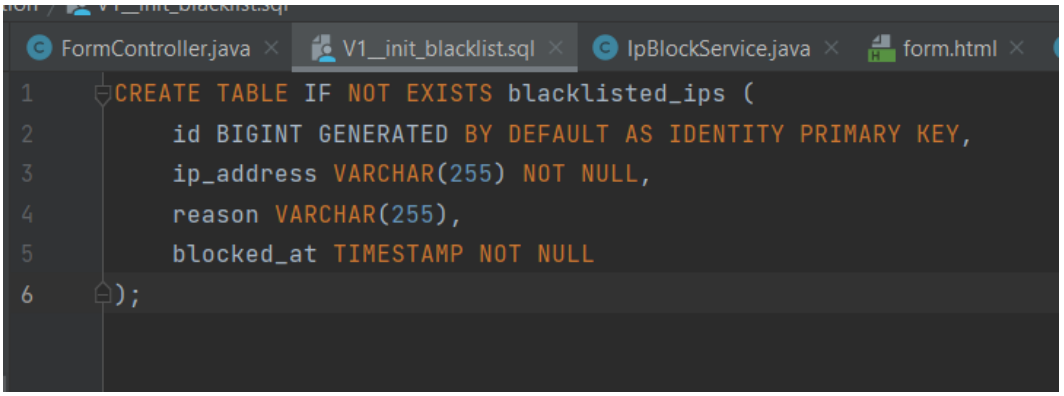


Рисунок 3.2 – Міграційний файл

Для доступу до даних використовується інтерфейс BlacklistedIpRepository (рис. 3.3), реалізований на основі Spring Data JPA. Інтерфейс розширює JpaRepository та надає метод пошуку за IP-адресою, що дозволяє швидко отримати інформацію про актуальні блокування. Додатково можуть бути використані методи для збереження нового запису або оновлення наявного при повторному блокуванні.

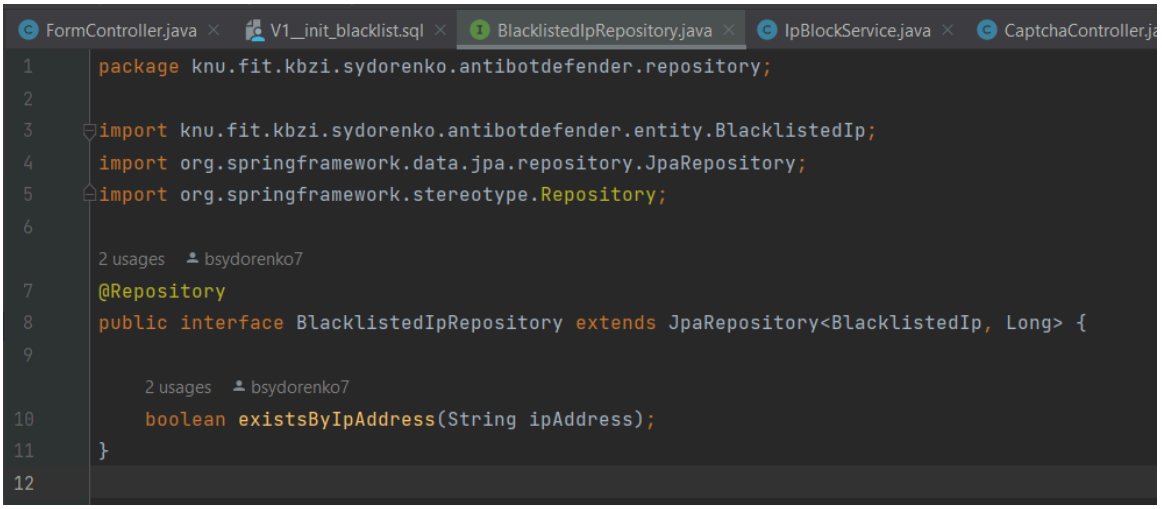


Рисунок 3.3 – Репозиторій BlacklistedIpRepository

Бізнес-логіка блокування реалізована в класі IpBlockService (рис. 3.4). Цей сервіс надає метод isBlacklisted(String ip), який перевіряє, чи є IP-адреса у чорному списку, та метод blockIfNotExists(String ip, String reason, String userAgent), який створює новий запис у базі, якщо IP ще не був заблокований. Сервіс виконує логування всіх подій блокування для подальшої аналітики.

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class IpBlockService {

    3 usages
    private final BlacklistedIpRepository repository;

    4 usages  bsydorenko7 *
    public void blockIfNotExists(String ipAddress, String reason) {
        if (!repository.existsByIpAddress(ipAddress)) {
            log.info("Blocking IP [{}] for reason: {}", ipAddress, reason);
            repository.save(
                    BlacklistedIp.builder().ipAddress(ipAddress).reason(reason).blockedAt(LocalDateTime.now()).build()
            );
        }
    }

    2 usages  bsydorenko7
    public boolean isBlacklisted(String ipAddress) {
        return repository.existsByIpAddress(ipAddress);
    }
}

```

Рисунок 3.4 – Сервіс IpBlockService

Важливою частиною механізму є фільтр RequestFilter (рис. 3.5), який інтегрується у Spring Boot через OncePerRequestFilter. На етапі preHandle перевіряється, чи поточна IP-адреса не входить до чорного списку. Якщо IP знайдено як активне блокування, запит не передається далі по ланцюгу обробки. Користувачу може бути повернута HTTP-відповідь з кодом 403 (Forbidden) або інша форма реакції залежно від конфігурації. Таким чином, усі підозрілі запити перехоплюються ще до потрапляння до контролерів або ресурсів системи.

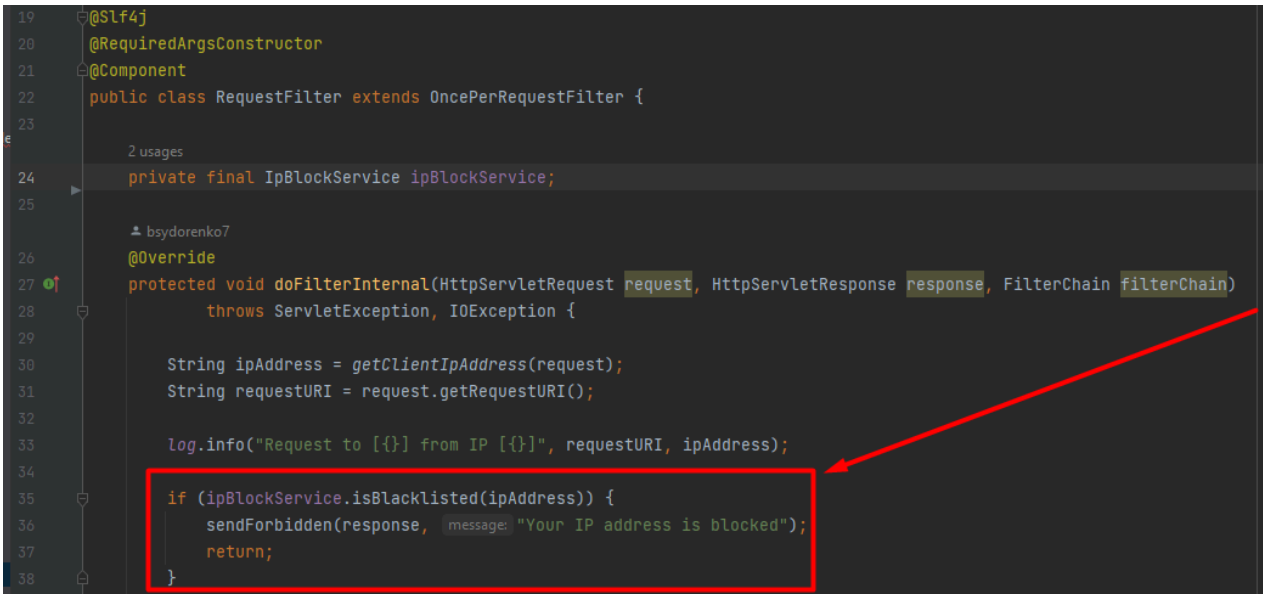


Рисунок 3.5 – RequestFilter з перевіркою на заблокований IP

Описана система забезпечує адаптивний, надійний і масштабований механізм блокування. Завдяки використанню сучасних інструментів JPA, Flyway та Spring Boot, логіка блокування є зрозумілою, гнучкою та готовою до розширення. Система блокування органічно інтегрується в загальну архітектуру захисту та дозволяє миттєво реагувати на бот-атаки або спроби обхідних маніпуляцій із боку зловмисників.

3.3 Застосування аналізу HTTP-запитів і перевірки заголовків

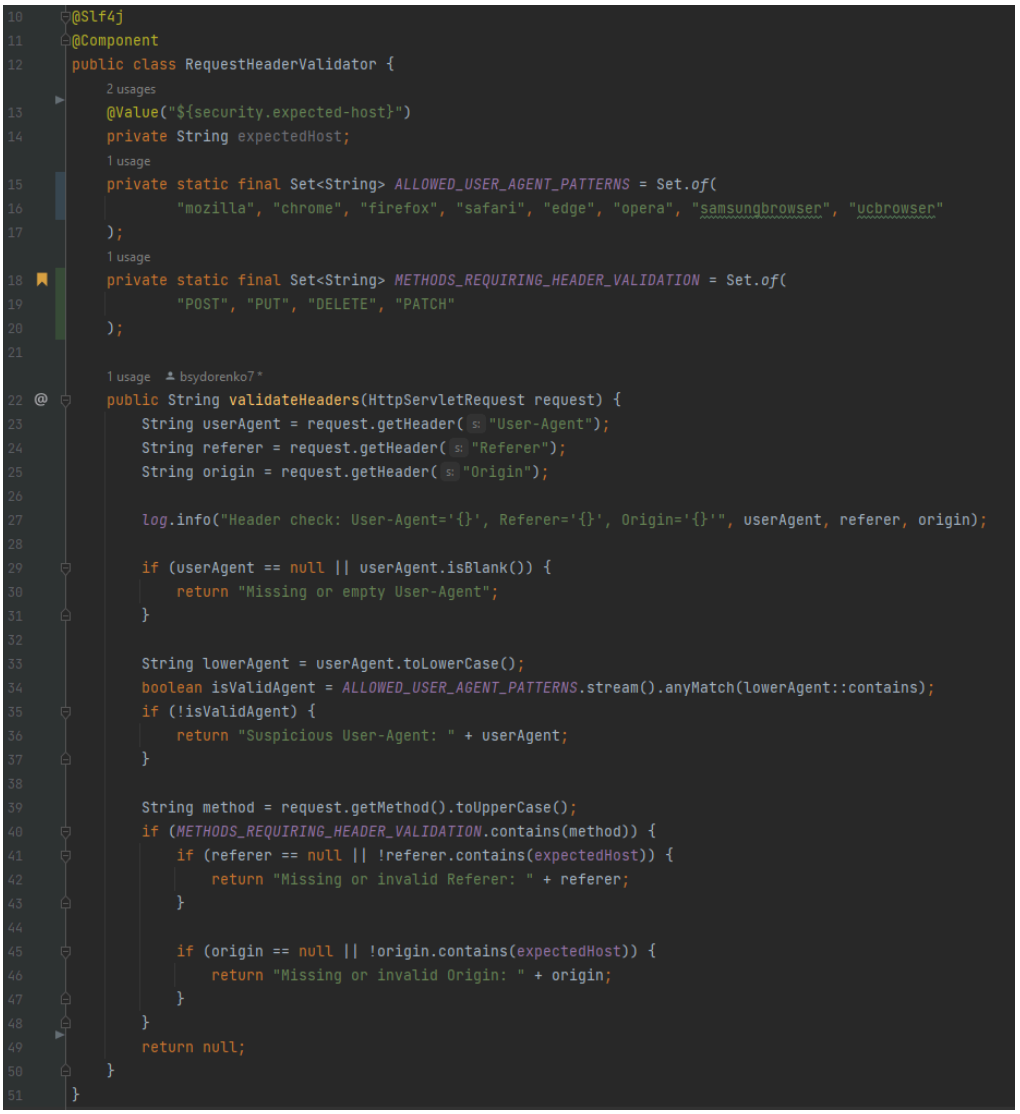
Одним із важливих аспектів аналізу підозрілої активності користувачів є перевірка структури та вмісту HTTP-запитів, зокрема – аналіз заголовків. HTTP-заголовки містять ключову інформацію про походження запиту, тип клієнта, його поведінку, а також взаємодію з веб-додатком. Саме тому механізм валідації заголовків є критично важливим для виявлення ботів або запитів, надісланих не через браузер.

Найважливішими з точки зору захисту є заголовки User-Agent, Referer та Origin. Заголовок User-Agent дозволяє визначити, який саме клієнт (браузер або інший HTTP-клієнт) виконав запит. Правильне значення зазвичай включає згадку про браузер, версію, операційну систему та іншу контекстну інформацію. Водночас,

у випадку ботів або скриптів, цей заголовок або відсутній, або містить нестандартні або підозрілі шаблони.

Заголовок Referer вказує на джерело, з якого здійснено запит. Він може містити URL сторінки, з якої користувач перейшов. У контексті певних типів запитів, таких як POST, PUT, PATCH або DELETE – відсутність Referer є вагомим індикатором того, що запит здійснено не через натискання посилання або дії користувача, а автоматично. Аналогічно, заголовок Origin містить домен походження запиту і дозволяє перевірити, чи був запит надісланий з очікуваного домену (тобто того, що належить серверу).

В нашій системі перевірка заголовків реалізована в окремому компоненті – RequestHeaderValidator (рис. 3.6). Цей клас є Spring-компонентом, який надає метод validateHeaders(HttpServletRequest request). Метод здійснює поетапну перевірку кожного із заголовків. По-перше, перевіряється наявність заголовка User-Agent. Якщо він відсутній або порожній – повертається причина блокування із зазначенням проблеми. Далі виконується пошук за дозволеним списком шаблонів, таких як «mozilla», «chrome», «firefox» тощо. Якщо жоден із шаблонів не знайдено у значенні User-Agent, запит вважається підозрілим. Наступним кроком виконується перевірка заголовків Referer та Origin для тих методів, які потребують суворої перевірки (POST, PUT, DELETE, PATCH). Значення цих заголовків повинні містити очікуване ім’я хоста, яке зчитується з конфігураційного параметра expected host. Якщо один із заголовків відсутній або не містить очікуваний домен, метод повертає відповідну причину для подальшого блокування.



```java
@Slf4j
@Component
public class RequestHeaderValidator {
    2 usages
    @Value("${security.expected-host}")
    private String expectedHost;
    1 usage
    private static final Set<String> ALLOWED_USER_AGENT_PATTERNS = Set.of(
            "mozilla", "chrome", "firefox", "safari", "edge", "opera", "samsungbrowser", "ucbrowser"
    );
    1 usage
    private static final Set<String> METHODS_REQUIRING_HEADER_VALIDATION = Set.of(
            "POST", "PUT", "DELETE", "PATCH"
    );

    1 usage  bsydorenko7 *
    public String validateHeaders(HttpServletRequest request) {
        String userAgent = request.getHeader( s: "User-Agent");
        String referer = request.getHeader( s: "Referer");
        String origin = request.getHeader( s: "Origin");

        log.info("Header check: User-Agent='{}', Referer='{}', Origin='{}'", userAgent, referer, origin);

        if (userAgent == null || userAgent.isBlank()) {
            return "Missing or empty User-Agent";
        }

        String lowerAgent = userAgent.toLowerCase();
        boolean isValidAgent = ALLOWED_USER_AGENT_PATTERNS.stream().anyMatch(lowerAgent::contains);
        if (!isValidAgent) {
            return "Suspicious User-Agent: " + userAgent;
        }

        String method = request.getMethod().toUpperCase();
        if (METHODS_REQUIRING_HEADER_VALIDATION.contains(method)) {
            if (referer == null || !referer.contains(expectedHost)) {
                return "Missing or invalid Referer: " + referer;
            }

            if (origin == null || !origin.contains(expectedHost)) {
                return "Missing or invalid Origin: " + origin;
            }
        }
        return null;
    }
}
```

Рисунок 3.6 – Компонент RequestHeaderValidator

Результатом роботи validateHeaders є текстова причина для блокування або null, якщо усі перевірки пройдено. Саме це значення потім використовується у фільтрі RequestFilter (рис. 3.7). Там викликається requestHeaderValidator.validateHeaders(request), і якщо результат не null – IP блокується викликом blockIfNotExists, а клієнту повертається відповідь 403 зі зрозумілою причиною блокування. Таким чином, заголовки є не лише індикаторами, а й джерелом детальної діагностики, яка дозволяє формувати гнучку політику блокування.

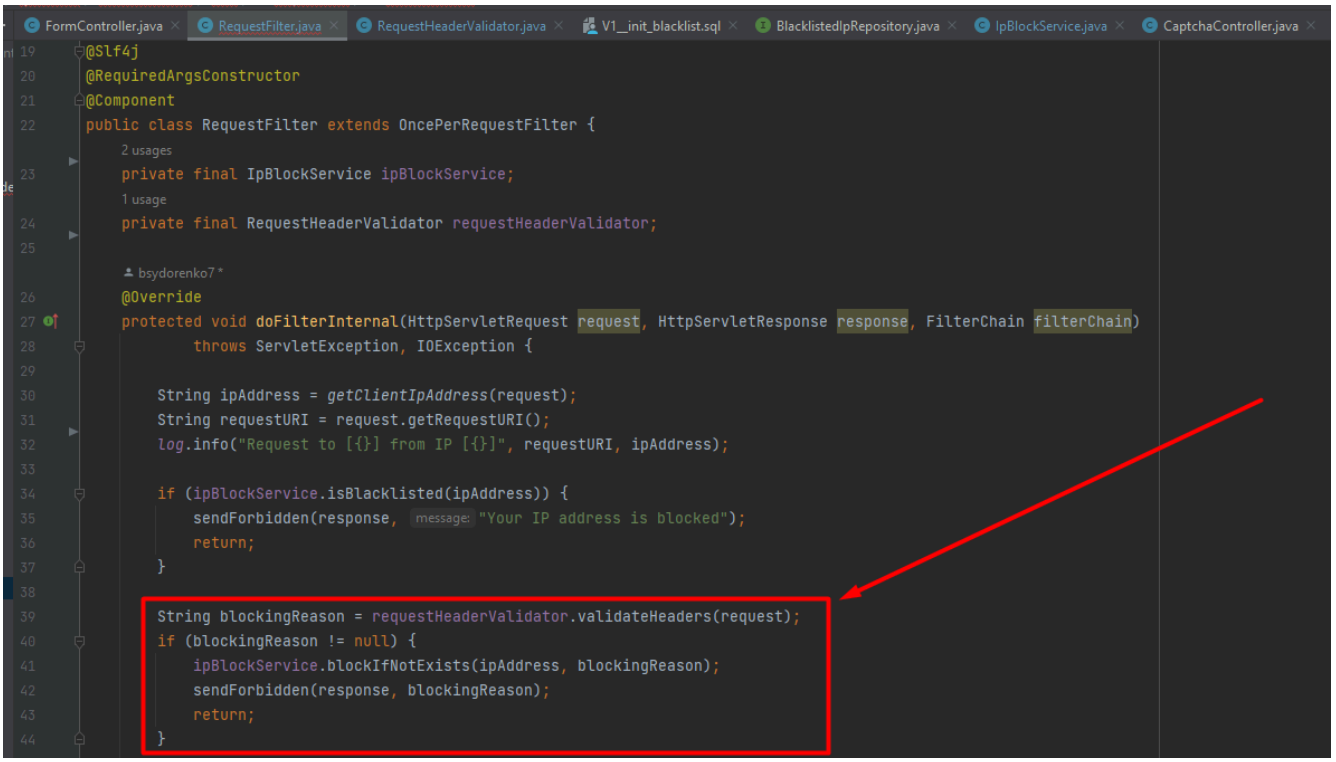


Рисунок 3.7 – RequestFilter з валідацією заголовків

Реалізація цього механізму дозволяє системі реагувати навіть на найменші відхилення від очікуваної поведінки HTTP-клієнта, що значно підвищує ефективність захисту без застосування надлишкових обчислювальних засобів. Крім того, кожен зафіксований інцидент супроводжується детальною причиною, яка зберігається в базі даних, що уможливлює подальший аналіз і вдосконалення правил валідації.

3.4 Побудова поведінкового і евристичного аналізу користувача

Один із найскладніших викликів у сфері веб-безпеки полягає у виявленні та фільтрації шкідливої активності, яка намагається імітувати легітимних користувачів. У таких випадках класичних перевірок заголовків HTTP часто буває недостатньо. Саме тому у сучасних системах захисту активно використовуються підходи поведінкового та евристичного аналізу, які дозволяють визначити, чи справді дію виконує людина, чи це результат автоматизованого процесу.

Поведінковий захист спирається на спостереження за реальними діями користувача у браузері. До таких дій можна віднести: рухи миші, натискання клавіш, взаємодію з елементами форми, переміщення фокусу, швидкість заповнення полів. Людина не може миттєво реагувати та виконувати всі ці дії з однаковою точністю й швидкістю щоразу. У протилежність цьому – бот, або headless-скрипт, діє миттєво, без затримок, не виконує рухів миші, ігнорує JavaScript-події. Саме ці характеристики дозволяють нам відрізняти справжнього користувача від автоматизованого агента.

Евристичний підхід у виявленні ботів ґрунтується на правилах, що сформовані шляхом аналізу типових шаблонів поведінки зловмисників. Це може бути: надмірна швидкість надсилання форм, заповнення прихованих полів, відсутність стандартних клієнтських атрибутів, повторювані дії, а також використання headless-браузерів або HTTP-клієнтів, які не проходять перевірки на поведінкові патерни. Таким чином, евристика дозволяє не лише фіксувати конкретні порушення, а й робити загальний висновок про ймовірність того, що дії виконуються ботом.

У виконаній реалізації було поєднано обидва підходи. Поведінковий та евристичний захист реалізовано через декілька незалежних механізмів: Behaviour Detector, Honeypot Mechanism та Form Timing Mechanism. Кожен із них функціонує автономно, але всі об'єднані у єдину систему прийняття рішення про блокування підозрілого IP. У разі спрацювання будь-якого з тригерів, IP користувача миттєво додається до чорного списку, що запобігає подальшій активності без необхідності ручного втручання.

Поведінковий аналіз реалізується через компонент Behaviour Detector, який складається з клієнтської та серверної частин. На стороні клієнта [Додаток Б] з використанням JavaScript відстежується наявність трьох основних ознак: чи було зафіксовано рух миші, чи були натискання клавіш, та чи браузер працює у повноцінному режимі. Для визначення цього використовується перевірка флагу navigator.webdriver. Вказані дані збираються під час заповнення форми й передаються на сервер через окремий POST-запит на ендпоінт /behavior.

На стороні сервера обробкою цього запиту займається клас BehaviorController (рис. 3.8), який отримує дані та виконує перевірку кожного з індикаторів. Якщо хоча б один із них свідчить про бот-активність (наприклад, відсутність мишкодії або натискань клавіш, або активований headless-режим), IP-адресу користувача негайно блокують методом blockIfNotExists, із збереженням причини «Suspicious behavior». У відповідь клієнт отримує повідомлення з HTTP-статусом 403 без можливості продовження взаємодії.

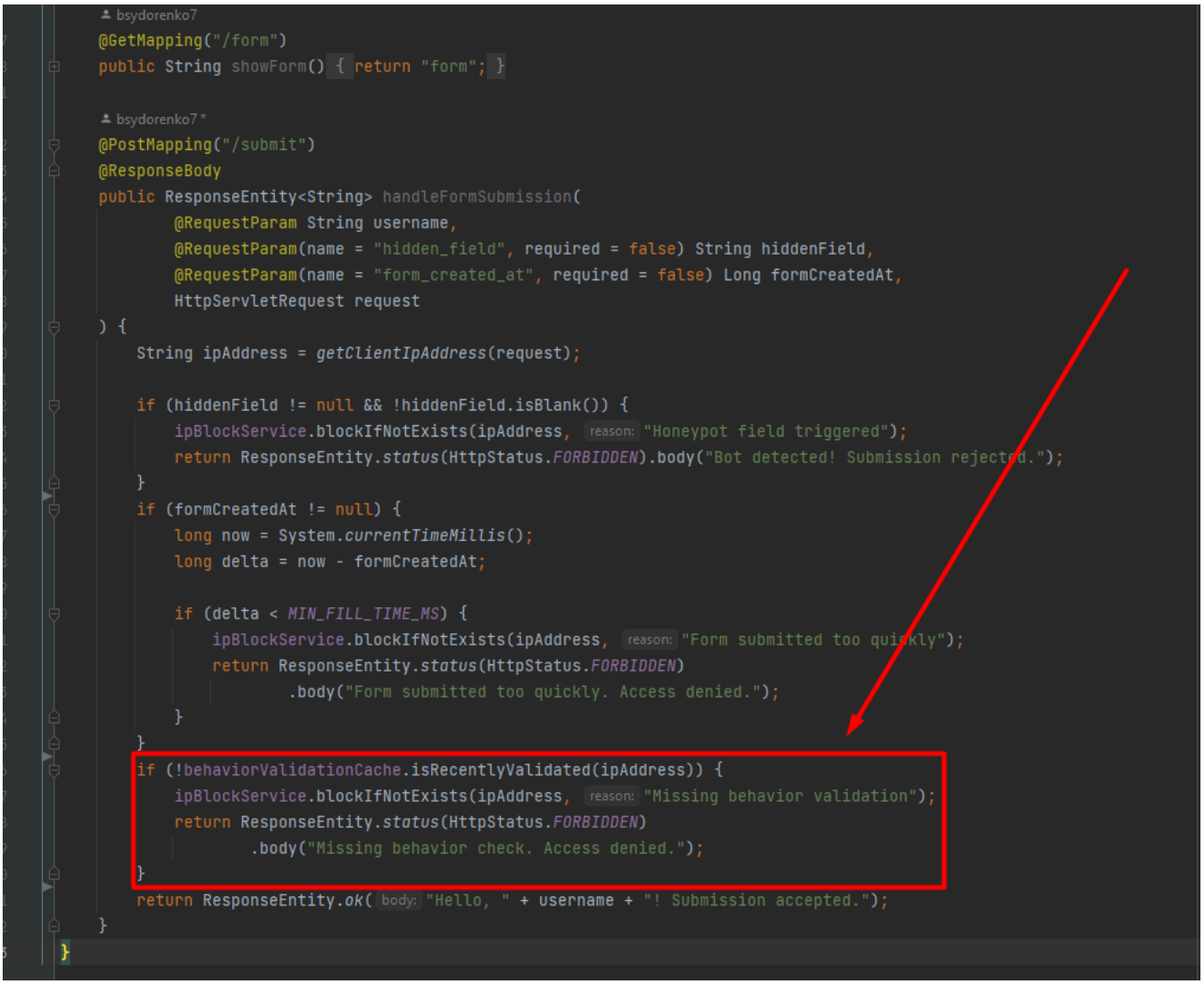
```
bsydorenko7 *
@Slf4j
@RestController
@RequiredArgsConstructor
public class BehaviorController {
    1 usage
    private final IpBlockService ipBlockService;
    1 usage
    private final BehaviorValidationCache behaviorValidationCache;

    bsydorenko7
    @PostMapping("/behavior")
    @ResponseBody
    public ResponseEntity<Void> handleBehavior(@RequestBody BehaviorPayload payload, HttpServletRequest request) {
        String ip = getClientIpAddress(request);

        if (payload.headless() || !payload.hasMouseMove() || !payload.hasKeyPress()) {
            ipBlockService.blockIfNotExists(ip, reason: "Suspicious behavior: " + payload);
            return ResponseEntity.status(403).build();
        }
        behaviorValidationCache.markValidated(ip);
        return ResponseEntity.ok().build();
    }

    1 usage  bsydorenko7
    public record BehaviorPayload(boolean hasMouseMove, boolean hasKeyPress, boolean headless) {}
}
```

Рисунок 3.8 – Контроллер BehaviorController

Після успішної перевірки поведінки IP-адреса користувача заноситься до короткотривалого кешу (BehaviorValidationCache) строком на 30 секунд (рис. 3.9). Це означає, що протягом цього часу клієнт вважається перевіреним і може виконувати подальші дії, зокрема надсилати форму.

Рисунок 3.9 – Кеш для перевірених IP BehaviorValidatorCache

Клас FormController, який відповідає за обробку форми (рис. 3.10), перевіряє, чи IP користувача був попередньо верифікований через поведінкову перевірку. Якщо запису у кеші немає – тобто клієнт не пройшов валідацію через /behavior – IP-адреса негайно блокується з причиною «Missing behavior validation», а відповідь серверу має статус HTTP 403. Лише після успішного проходження поведінкового аналізу система дозволяє користувачу виконати запит на сабміт форми.



```
bsydorenko7
@GetMapping("/form")
public String showForm() { return "form"; }

bsydorenko7 *
@PostMapping("/submit")
@ResponseBody
public ResponseEntity<String> handleFormSubmission(
        @RequestParam String username,
        @RequestParam(name = "hidden_field", required = false) String hiddenField,
        @RequestParam(name = "form_created_at", required = false) Long formCreatedAt,
        HttpServletRequest request
) {
    String ipAddress = getClientIpAddress(request);

    if (hiddenField != null && !hiddenField.isBlank()) {
        ipBlockService.blockIfNotExists(ipAddress, reason: "Honeypot field triggered");
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body("Bot detected! Submission rejected.");
    }
    if (formCreatedAt != null) {
        long now = System.currentTimeMillis();
        long delta = now - formCreatedAt;

        if (delta < MIN_FILL_TIME_MS) {
            ipBlockService.blockIfNotExists(ipAddress, reason: "Form submitted too quickly");
            return ResponseEntity.status(HttpStatus.FORBIDDEN)
                    .body("Form submitted too quickly. Access denied.");
        }
    }
    if (!behaviorValidationCache.isRecentlyValidated(ipAddress)) {
        ipBlockService.blockIfNotExists(ipAddress, reason: "Missing behavior validation");
        return ResponseEntity.status(HttpStatus.FORBIDDEN)
                .body("Missing behavior check. Access denied.");
    }
    return ResponseEntity.ok( body: "Hello, " + username + "! Submission accepted.");
}
}
```

Рисунок 3.10 – Контроллер FormController з behavior перевіркою

Таким чином, впроваджено обов’язкову поведінкову перевірку як передумову для будь-якої подальшої взаємодії з веб-сервісом. Такий підхід дозволяє ефективно виявляти headless-клієнтів і ботів, які не здатні взаємодіяти з фронтендом згідно з очікуваними сценаріями.

Ще одним прикладом застосування евристичного підходу є механізм honeypot – приховані поля у формах, які не видно справжньому користувачеві, але які можуть бути заповнені ботом через незнання або скриптову обробку всіх інпутів. Ці поля мають стилі «display: none» або «visibility: hidden» [Додаток Б], або не мають фокусу у навігаційній послідовності. Якщо такі поля заповнені – це явна ознака того, що взаємодія здійснюється не людиною, а автоматизованим агентом. На стороні сервера перевірка honeypot реалізована у FormController (рис. 3.11). Під час обробки

POST-запиту сервер перевіряє значення прихованого поля «hidden_field», і якщо воно не пусте – IP блокується з причиною «Honeypot field triggered». Такий підхід є простим у реалізації та майже не дає хибнопозитивних результатів.

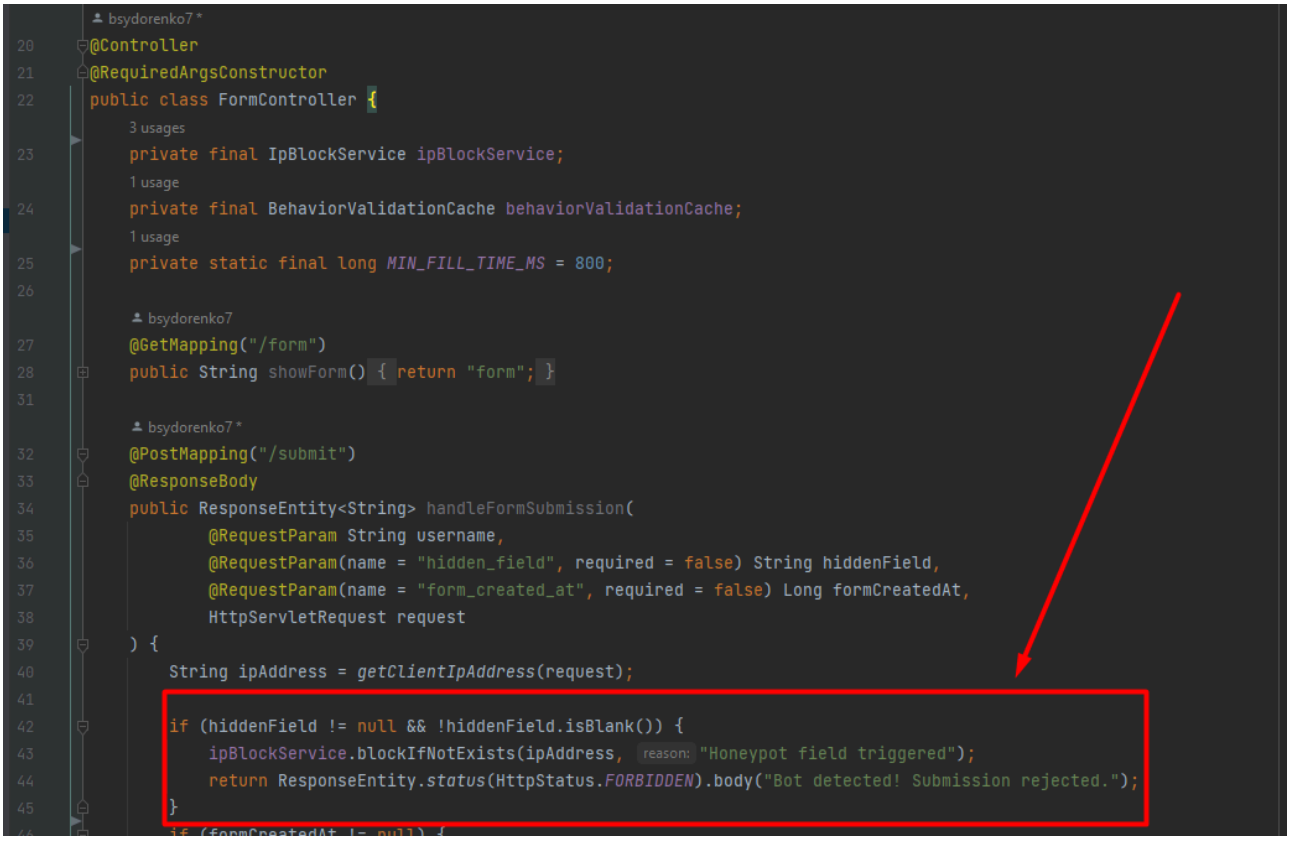
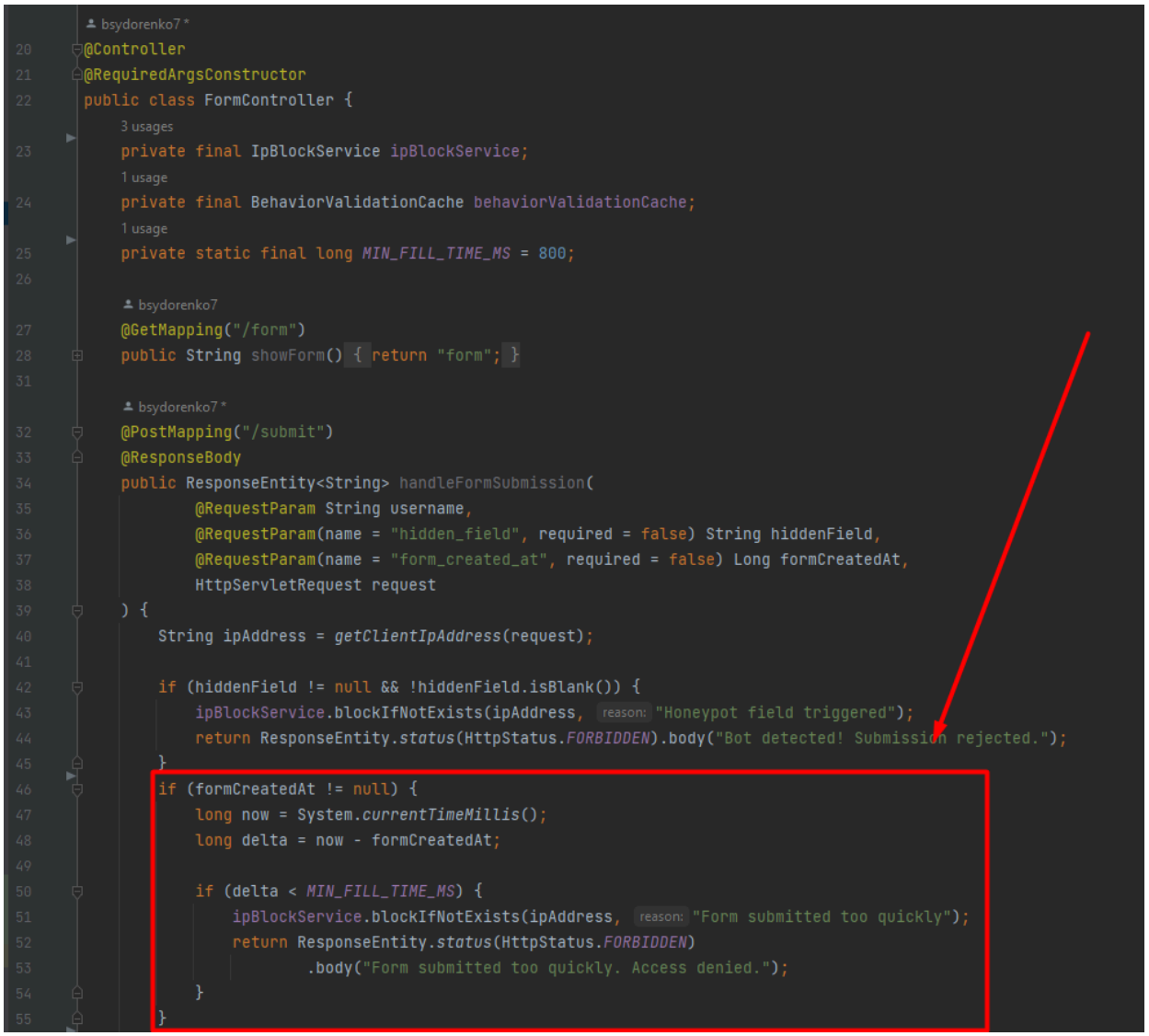


Рисунок 3.11 – Контроллер FormController з honeypot перевіркою

Третім компонентом системи захисту є Form Timing Mechanism. Він ґрунтується на відстеженні часу, за який була заповнена форма. Після завантаження сторінки JavaScript вставляє поточний час у мілісекундах у приховане поле «form_created_at» [Додаток Б]. Під час надсилання форми бекенд (рис. 3.12) обраховує різницю між поточним часом та переданим значенням. Якщо форма була заповнена менш ніж за 800 мс – це вважається неприродною швидкістю заповнення, характерною для ботів. У цьому випадку знову викликається blockIfNotExists і формується повідомлення «Form submitted too quickly» із зазначенням затраченого часу. Це дозволяє виявляти найпростіші скрипти, які не чекають та ігнорують затримки.



```
bsydorenko7 *
@Controller
@RequiredArgsConstructor
public class FormController {
    3 usages
    private final IpBlockService ipBlockService;
    1 usage
    private final BehaviorValidationCache behaviorValidationCache;
    1 usage
    private static final long MIN_FILL_TIME_MS = 800;

    bsydorenko7
    @GetMapping("/form")
    public String showForm() { return "form"; }

    bsydorenko7 *
    @PostMapping("/submit")
    @ResponseBody
    public ResponseEntity<String> handleFormSubmission(
            @RequestParam String username,
            @RequestParam(name = "hidden_field", required = false) String hiddenField,
            @RequestParam(name = "form_created_at", required = false) Long formCreatedAt,
            HttpServletRequest request
    ) {
        String ipAddress = getClientIpAddress(request);

        if (hiddenField != null && !hiddenField.isBlank()) {
            ipBlockService.blockIfNotExists(ipAddress, reason: "Honeypot field triggered");
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body("Bot detected! Submission rejected.");
        }
        if (formCreatedAt != null) {
            long now = System.currentTimeMillis();
            long delta = now - formCreatedAt;

            if (delta < MIN_FILL_TIME_MS) {
                ipBlockService.blockIfNotExists(ipAddress, reason: "Form submitted too quickly");
                return ResponseEntity.status(HttpStatus.FORBIDDEN)
                        .body("Form submitted too quickly. Access denied.");
            }
        }
```

Рисунок 3.12 – Контроллер FormController з Form Timing Mechanism

Таким чином, кожен із трьох механізмів – поведінковий, honeypot та таймінг – забезпечує свою ділянку захисту. У сукупності вони дозволяють ефективно фільтрувати запити ще до моменту взаємодії з основною логікою веб-додатку. Усі тригерні дії ведуть до централізованого методу блокування IP, з фіксацією причини та можливістю подальшого аналізу або розблокування адміністративно. При цьому система не створює додаткового навантаження на сервер та є повністю прозорою для кінцевого користувача, який у випадку легітимної поведінки взагалі не помітить жодної додаткової перевірки.

3.5 Інтеграція адаптивного захисту з застосуванням CAPTCHA

Важливою частиною побудови багаторівневого захисту веб-додатку є адаптивна реакція на підозрілу активність. Одним з основних індикаторів такої активності є частота запитів з однієї IP-адреси. Щоб виявляти користувачів або ботів, які надсилають надмірну кількість запитів за короткий проміжок часу, у системі реалізовано компонент RateLimiterService (рис. 3.13).

```
2 usages  bsydorenko7 *
@Service
public class RateLimiterService {
    1 usage
    private static final int MAX_REQUESTS = 10;
    1 usage
    private static final long TIME_WINDOW_MILLIS = 10_000;

    1 usage
    private final Cache<String, Deque<Long>> requestCache = Caffeine.newBuilder()
            .expireAfterAccess( duration: 5, TimeUnit.MINUTES)
            .maximumSize(10_000)
            .build();

    1 usage  bsydorenko7 *
    public boolean isAllowed(String ip) {
        long now = Instant.now().toEpochMilli();
        Deque<Long> timestamps = requestCache.get(ip, k -> new ArrayDeque<>());
        synchronized (timestamps) {
            while (!timestamps.isEmpty() && now - timestamps.peekFirst() > TIME_WINDOW_MILLIS) {
                timestamps.pollFirst();
            }

            if (timestamps.size() >= MAX_REQUESTS) {
                return false;
            }

            timestamps.addLast(now);
            return true;
        }
    }
}
```

Рисунок 3.13 – Сервіс RateLimiterService

RateLimiterService є сервісом, який веде облік запитів із кожної IP-адреси та перевіряє, чи не перевищено поріг допустимої частоти викликів. Якщо IP-адреса намагається здійснити більше запитів, ніж дозволено політикою, система вважає її потенційно шкідливою. Однак, на відміну від класичних підходів, які одразу блокують IP, наше рішення діє м’якше – виконується перенаправлення на CAPTCHA.

Технічно RateLimiterService побудовано на основі бібліотеки Caffeine, яка забезпечує високошвидкісний кеш із підтримкою TTL. Для кожної IP-адреси ведеться лічильник кількості запитів за останній період. Якщо кількість перевищує порогове значення, метод isAllowed(ip) повертає false. У цьому випадку користувач не отримує відмову, а спрямовується на CAPTCHA, яка дозволяє розмежувати справжніх користувачів та ботів без створення незручностей.

У фільтрі RequestFilter (рис. 3.14) ця логіка виглядає наступним чином: викликається rateLimiterService.isAllowed(ipAddress). Якщо IP перевищив поріг, виконується перенаправлення на спеціальний маршрут /captcha, передаючи IP і причину «Rate limit exceeded». Це дозволяє уникнути помилкових блокувань і надає користувачу шанс підтвердити легітимність.

```java
@Slf4j
@RequiredArgsConstructor
@Component
public class RequestFilter extends OncePerRequestFilter {

    2 usages
    private final IpBlockService ipBlockService;
    1 usage
    private final RequestHeaderValidator requestHeaderValidator;
    1 usage
    private final RateLimiterService rateLimiterService;

    bsydorenko7 *
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        String ipAddress = getClientIpAddress(request);
        String requestURI = request.getRequestURI();
        log.info("Request to [{}] from IP [{}]", requestURI, ipAddress);

        if (ipBlockService.isBlacklisted(ipAddress)) {
            sendForbidden(response, message: "Your IP address is blocked");
            return;
        }
        String blockingReason = requestHeaderValidator.validateHeaders(request);
        if (blockingReason != null) {
            ipBlockService.blockIfNotExists(ipAddress, blockingReason);
            sendForbidden(response, blockingReason);
            return;
        }
        if (!rateLimiterService.isAllowed(ipAddress)) {
            redirectToCaptcha(response, ipAddress, reason: "Rate limit exceeded");
            return;
        }

        filterChain.doFilter(request, response);
    }
```

Рисунок 3.14 – RequestFilter з вбудованою перевірка RateLimiterService та перенаправленням на CAPTCHA

CAPTCHA в системі Anti Bot Defender виконує роль фільтра другого рівня. Вона активується лише у випадках, коли поведінка користувача викликає підозру: перевищено ліміт запитів, некоректні заголовки або підозріла поведінка. Такий підхід дозволяє зберегти зручність використання сервісу більшістю користувачів, залишаючи жорсткі перевірки лише для потенційно небезпечного трафіку.

Першим кроком для реалізації CAPTCHA є реєстрація сайту для Google reCAPTCHA, який виконується через адміністративну панель (рис. 3.15). Для проєкту використовується тип CAPTCHA – версія 2, тип «прапорець», із дозволеними доменами: «localhost», «127.0.0.1», «antibot.fit.knu.ua.test».

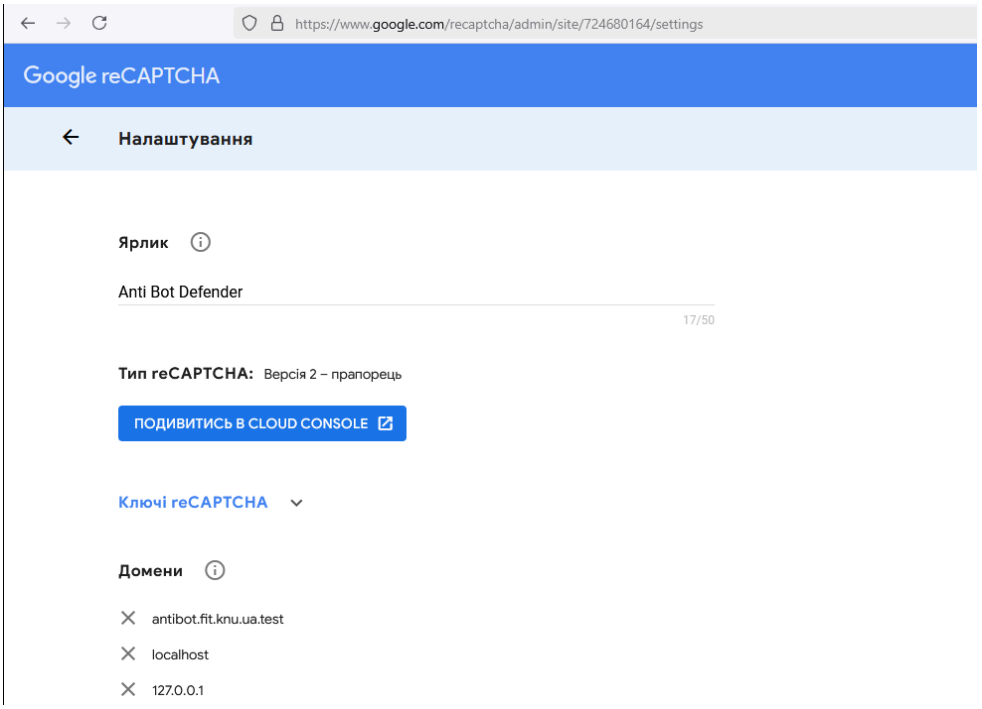


Рисунок 3.15 – Адміністративна панель Google reCaptcha

Сторінка з CAPTCHA реалізована як Thymeleaf-шаблон captcha.html (рис. 3.16). Вона містить поля для передачі IP-адреси, причини блокування та reCAPTCHA site key, а також саму CAPTCHA-форму, що вставляється через офіційний «<script>» від Google.

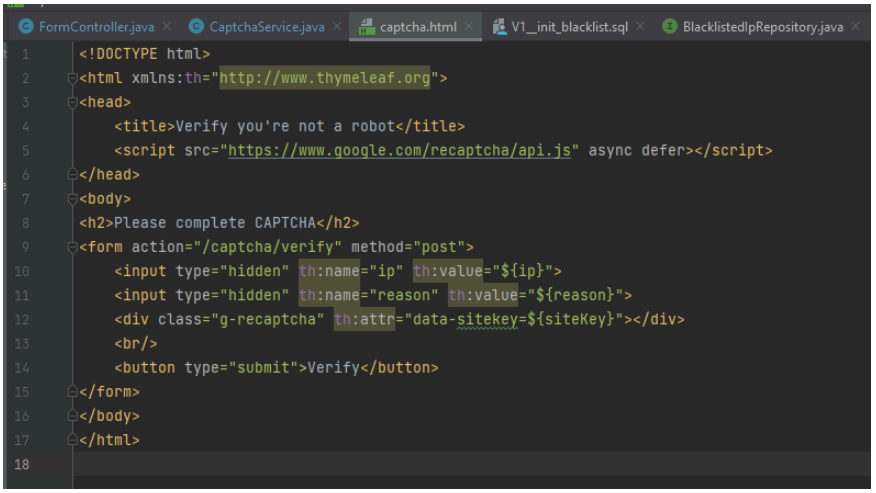
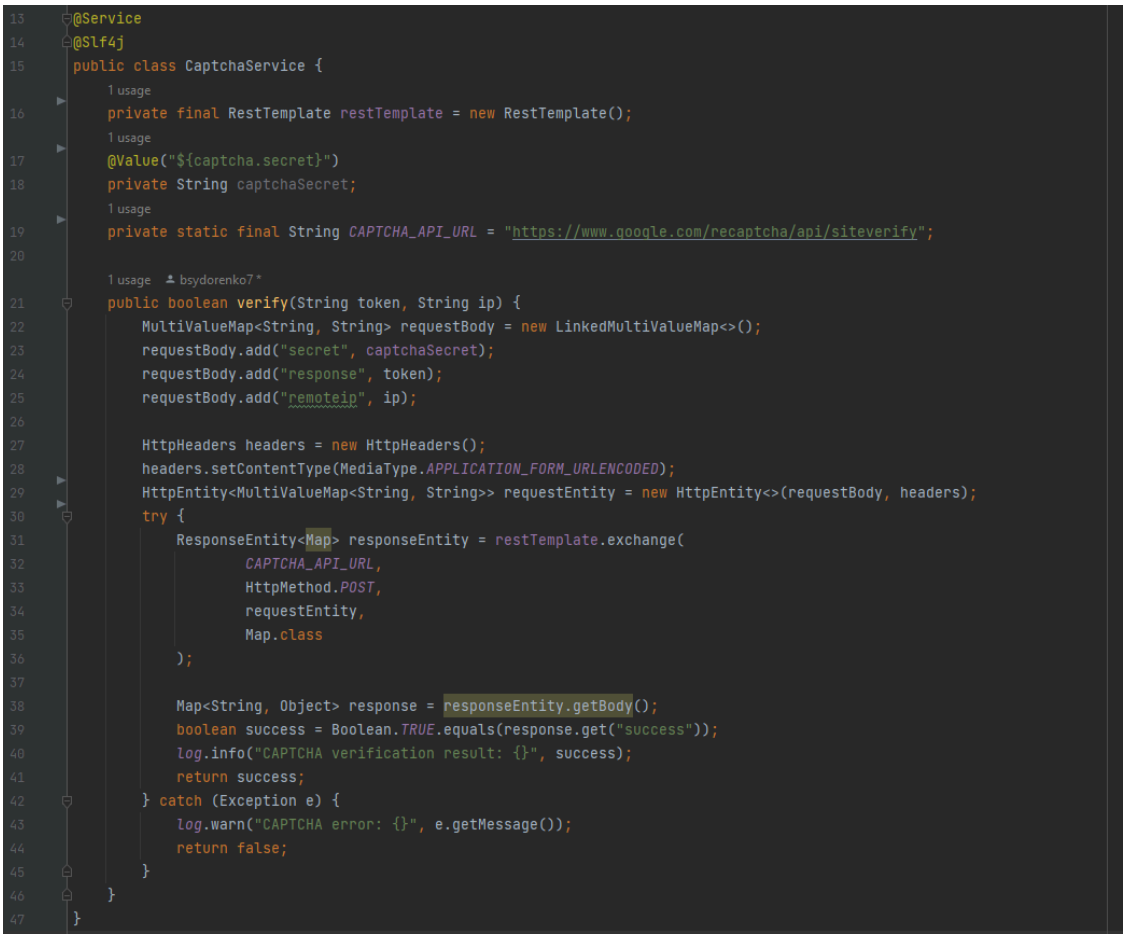


Рисунок 3.16 – Файл captcha.html

Верифікація CAPTCHA виконується у CaptchaService (рис. 3.17). Сервіс формує запит до Google API /siteverify, передає токен, секретний ключ і IP-адресу, і очікує JSON-відповідь і повертає її з методу verify, true – якщо верифікація пройшла успішго та false, якщо ні.



```java
@Service
@Slf4j
public class CaptchaService {
    1 usage
    private final RestTemplate restTemplate = new RestTemplate();
    1 usage
    @Value("${captcha.secret}")
    private String captchaSecret;
    1 usage
    private static final String CAPTCHA_API_URL = "https://www.google.com/recaptcha/api/siteverify";

    1 usage  bsydorenko7 *
    public boolean verify(String token, String ip) {
        MultiValueMap<String, String> requestBody = new LinkedMultiValueMap<>();
        requestBody.add("secret", captchaSecret);
        requestBody.add("response", token);
        requestBody.add("remoteip", ip);

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        HttpEntity<MultiValueMap<String, String>> requestEntity = new HttpEntity<>(requestBody, headers);
        try {
            ResponseEntity<Map> responseEntity = restTemplate.exchange(
                    CAPTCHA_API_URL,
                    HttpMethod.POST,
                    requestEntity,
                    Map.class
            );

            Map<String, Object> response = responseEntity.getBody();
            boolean success = Boolean.TRUE.equals(response.get("success"));
            log.info("CAPTCHA verification result: {}", success);
            return success;
        } catch (Exception e) {
            log.warn("CAPTCHA error: {}", e.getMessage());
            return false;
        }
    }
}
```

Рисунок 3.17 – Сервіс CaptchaService

Контролер CaptchaController (рис. 3.18) забезпечує логіку відображення та перевірки CAPTCHA. Він отримує параметри з фільтру, відображає сторінку користувачу, а потім приймає результат введення через маршрут /captcha/verify. У цьому запиті передається токен reCAPTCHA, а також IP і причина, які зберігаються протягом перевірки. Якщо CAPTCHA не пройдено, IP остаточно блокується методом ipBlockService.blockIfNotExists з відповідною причиною.

```java
bsydorenko7 *
@Controller
@RequiredArgsConstructor
public class CaptchaController {
    1 usage
    private final IpBlockService ipBlockService;
    1 usage
    private final CaptchaService captchaService;
    1 usage
    @Value("${captcha.site-key}")
    private String captchaSiteKey;

    bsydorenko7
    @GetMapping("/captcha")
    public String showCaptchaForm(Model model, @RequestParam String ip, @RequestParam String reason) {
        model.addAttribute( attributeName: "ip", ip);
        model.addAttribute( attributeName: "reason", reason);
        model.addAttribute( attributeName: "siteKey", captchaSiteKey);
        return "captcha";
    }

    bsydorenko7
    @PostMapping("/captcha/verify")
    public String verifyCaptcha(
            @RequestParam String ip,
            @RequestParam String reason,
            @RequestParam("g-recaptcha-response") String captchaToken
    ) {
        if (!captchaService.verify(captchaToken, ip)) {
            ipBlockService.blockIfNotExists(ip, reason);
        }
        return "redirect:/form";
    }
}
```

Рисунок 3.18 – Контроллер CaptchaController

Таким чином, компонент RateLimiterService є першим етапом адаптивного захисту: він виявляє підозрілі IP за частотою запитів і спрямовує їх на перевірку через CAPTCHA. Успішне проходження CAPTCHA дозволяє користувачу продовжити роботу без блокування, а провал – призводить до внесення IP до чорного списку. Така багаторівнева архітектура дозволяє забезпечити високу точність виявлення ботів без надмірного тиску на легітимних користувачів.

Висновки за розділом 3

Підсумовуючи, було детально досліджено та реалізовано багаторівневу систему захисту Java веб-додатку від шкідливої автоматизованої активності.

Запропонована архітектура поєднує як класичні механізми перевірки, так і сучасні поведінкові й евристичні підходи, а також адаптивні механізми із застосуванням CAPTCHA. Кожен компонент системи виконує свою вузькоспеціалізовану функцію, однак у сукупності вони утворюють потужний бар'єр, який дозволяє ефективно виявляти й нейтралізовувати бот-атаку на ранніх етапах обробки HTTP-запиту.

Розділ розпочався з побудови архітектури рішення та вибору технологій. Було обґрунтовано використання Spring Boot як основи проєкту, Thymeleaf для побудови клієнтської частини, Spring Security для базової безпеки, а також Flyway, H2 та Caffeine як ефективні інструменти для збереження даних, міграцій та кешування відповідно. Побудовано чітку структуру, де IP-адреса перевіряється ще до надходження запиту до контролера, і залежно від результатів – блокується або обробляється далі.

Було реалізовано централізований механізм блокування IP-адрес з обґрунтованим зберіганням причин блокування. Кожен заблокований IP може бути проаналізований згодом, що підвищує контрольованість та прозорість системи. Реалізація бази даних, Flyway-міграцій, сервісів та репозиторіїв продемонструвала хорошу масштабованість та адаптивність рішення.

Особливу увагу в розділі було приділено аналізу HTTP-запитів і заголовків. Було обґрунтовано вибір User-Agent, Referer та Origin як основних полів для перевірки. Запропонований механізм RequestHeaderValidator забезпечує гнучке та масштабоване середовище для детекції аномалій, що можуть свідчити про бот-активність. Усі виявлені відхилення фіксуються у вигляді причини та передаються в IpBlockService для реакції.

Ще одним важливим кроком стало впровадження поведінкового та евристичного захисту. Визначення headless-браузерів, перевірка JavaScript-подій (рухи миші, клавіатура), аналіз часу заповнення форми та перевірка honeypot-поля дозволили досягти вищого рівня точності при виявленні автоматизованих загроз. Особливістю рішення є те, що перевірка відбувається до надсилання форми, і рішення приймається миттєво. Таким чином, бот навіть не отримує доступ до сервісної логіки.

Завершальним етапом архітектури став адаптивний захист із застосуванням CAPTCHA. Замість негайного блокування підозрілих запитів, система направляє користувача на CAPTCHA-сторінку, що дозволяє знизити ймовірність хибнопозитивних спрацювань. Якщо перевірку пройдено – користувач повертається до роботи, якщо ж ні – IP блокується. Реалізація включає в себе повну інтеграцію з Google reCAPTCHA v2, перевірку результатів через зовнішнє API та відповідну реакцію в системі. Такий підхід забезпечує надійний бар'єр для зловмисників, при цьому залишаючи можливість продовжити роботу легітимним користувачам.

Загалом, у третьому розділі сформовано повноцінну інфраструктуру, яка дозволяє виявляти, класифікувати та ізолювати шкідливу активність без шкоди для звичайного трафіку. Комбінація перевірок на різних рівнях (заголовки, частота, поведінка, CAPTCHA) дає змогу створити високоефективну антибот-платформу, придатну до масштабування та впровадження в реальні веб-сервіси різного рівня складності.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ПЕРЕВІРКА ПРАЦЕЗДАТНОСТІ РЕАЛІЗОВАНОГО МЕТОДУ ЗАХИСТУ

4.1 Емуляція бот-активності для перевірки маханізмів захисту

Метою тестування є перевірка спрацювання захисних механізмів у ситуаціях, які імітують поведінку ботів або підозрілих користувачів. Кожен сценарій містить опис дій, очікувану поведінку системи, команду для запуску, а також результат, який було зафіксовано в логах та в базі даних.

Цей сценарій перевіряє роботу механізму RequestHeaderValidator у випадку, коли клієнт надсилає запит без заголовка User-Agent. Такий тип запитів часто характерний для скриптів, утиліт або ботів, які не імітують браузер. Для перевірки було виконано HTTPS-запит до сторінки /form із порожнім заголовком User-Agent. Команда виконання запиту (рис. 4.1) виглядала так:

curl -k https://antibot.fit.knu.ua.test:8443/form -H "User-Agent:"

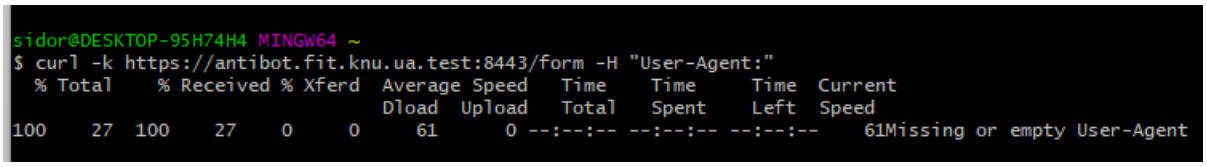


Рисунок 4.1 – Команда виконання запиту без User-Agent заголовку

У результаті система повернула повідомлення «Missing or empty User-Agent», і IP-адресу клієнта було автоматично заблоковано. Це підтверджується в логах застосунку (рис. 4.2).

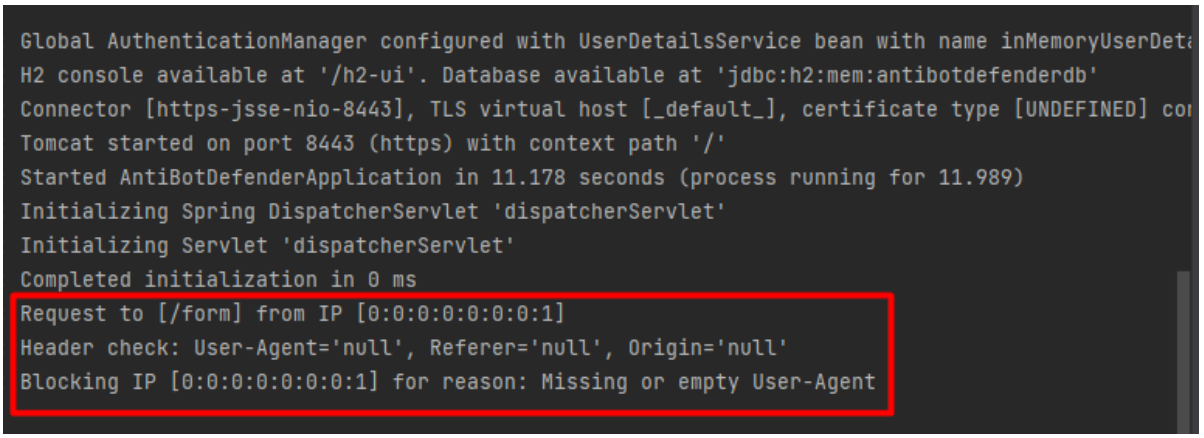


Рисунок 4.2 – Логи з блокуванням IP

А також у базі даних H2, де був створений відповідний запис у таблиці BLACKLISTED_IP (рис. 4.3) з IP-адресою та причиною блокування.

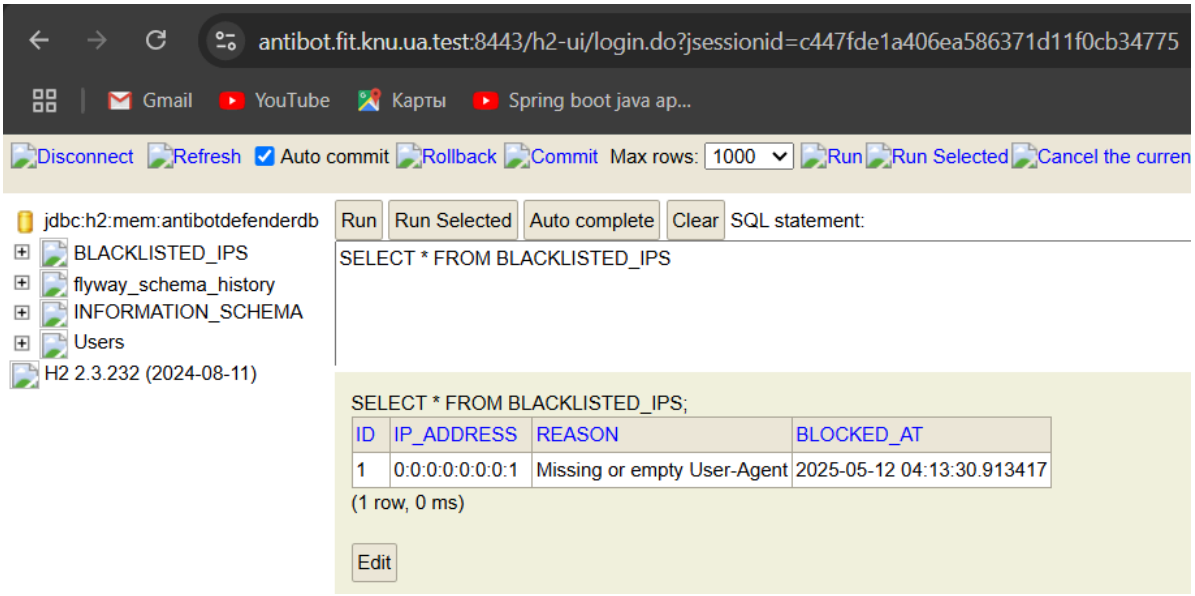


Рисунок 4.3 – Таблиця в базі данних з заблокованим IP

При повторному зверненні до /form через браузер користувач одразу отримував відповідь HTTP 403 з повідомленням «Your IP address is blocked» (рис. 4.4). Це підтверджує, що не лише механізм перевірки заголовків працює правильно, а й механізм блокування IP-адрес, реалізований у IpBlockService, функціонує відповідно до очікувань. Таким чином, захист активується неодноразово для одного й того самого IP, якщо його поведінка залишаєтсья підозрілою.

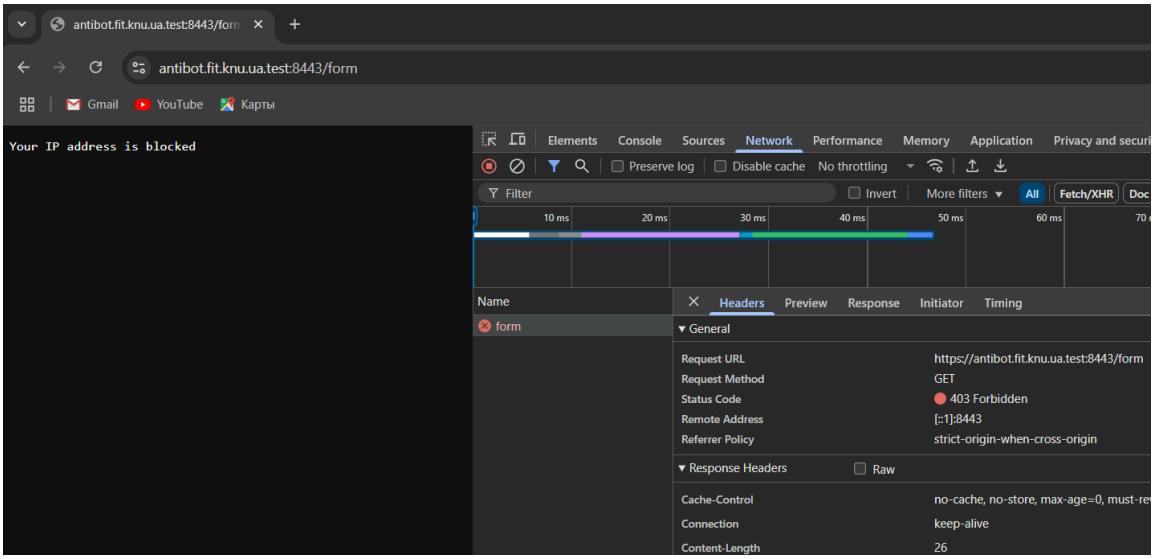


Рисунок 4.4 – Помилка 403 при повторному звернені з заблокованого клієнта

У другому сценарії перевіряється поведінка системи у випадку, коли запит містить підроблений або некоректний заголовок Referer. Цей заголовок, як правило, вказує на URL-адресу джерела переходу (тобто сторінку, з якої користувач натиснув посилання). Для захищених форм очікується, що Referer міститиме домен, якому довіряє сервер – у даному випадку https://antibot.fit.knu.ua.test:8443. У тесті було змодельовано ситуацію, коли бот або сторонній скрипт намагається надіслати запит з Referer, який не відповідає очікуваному (рис. 4.5).Для цього був використаний такий curl-запит:

```
curl -k -X POST https://antibot.fit.knu.ua.test:8443/form \
    -H "User-Agent: Mozilla/5.0" \
    -H "Referer: http://evil.com" \
    -H "Origin: https://antibot.fit.knu.ua.test:8443"
```

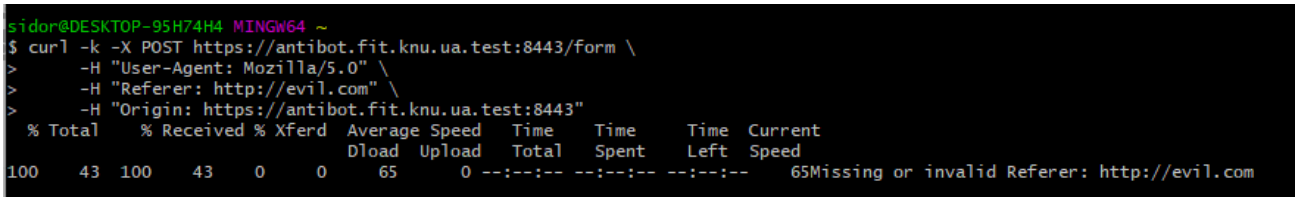


Рисунок 4.5 – Команда виконання запиту з некорректним Refer заголовком

Сервер виявив невідповідність Referer очікуваному домену і, згідно з логікою RequestHeaderValidator, згенерував повідомлення про підозрілий заголовок. У результаті IP клієнта було заблоковано через IpBlockService, що підтверджується як логами застосунку, так і фактом повернення відповіді HTTP 403 Forbidden при наступних зверненнях.

У третьому сценарії досліджується реакція системи захисту у випадку, коли запит не містить заголовка Origin. Цей заголовок зазвичай присутній у запитах, ініційованих браузером, особливо при POST- або CORS-запитах, і вказує на джерело, з якого був відправлений запит. У контексті безпеки, відсутність Origin може свідчити про ручне або скриптове формування запиту, що є типовою поведінкою ботів або зловмисних програм.Щоб змоделювати таку ситуацію, було сформовано запит, у якому Origin свідомо не передавався, а Referer залишався валідним:

```
curl -k -X POST https://antibot.fit.knu.ua.test:8443/form \
    -H "User-Agent: Mozilla/5.0" \
    -H "Referer: https://antibot.fit.knu.ua.test:8443"
```

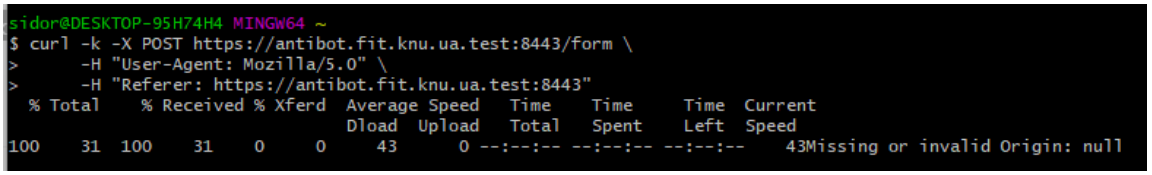


Рисунок 4.6 – Команда виконання запиту без передачі Origin заголовка

Механізм RequestHeaderValidator виявив відсутність Origin, оскільки тип запиту вимагав його наявність. У відповідь система класифікувала запит як потенційно шкідливий і активувала захисну реакцію – IP-адресу було негайно заблоковано з причиною «Missing or invalid Origin».

Після блокування, будь-які подальші запити з цієї IP-адреси незалежно від заголовків призводили до повернення HTTP 403 з повідомленням «Your IP address is blocked», що було зафіксовано як у веб-інтерфейсі, так і в логах застосунку. Це

демонструє, що система коректно виявляє ситуації з відсутнім критично важливим заголовком і блокує доступ до сервісу ще до моменту обробки запиту у контролері.

Наступний кейс спрямований на перевірку ситуації, коли клієнт не пройшов перевірку поведінки, тобто не викликав маршрут /behavior до сабміту форми. Така ситуація є типовою для headless-ботів, які не виконують JavaScript-логіку, що вбудована на фронтенді сайту. Згідно з реалізацією системи, в таких випадках IP-адреса має бути заблокована, оскільки вважається, що клієнт навмисно або технічно не здатен пройти базову перевірку інтеракції з інтерфейсом.

Щоб відтворити цей сценарій, був створений скрипт на основі Puppeteer (рис. 4.7) – headless-браузера, який автоматично відкриває сторінку /form, але не виконує жодної поведінкової взаємодії. У скрипті безпосередньо викликається сабміт форми без попереднього виклику /behavior.

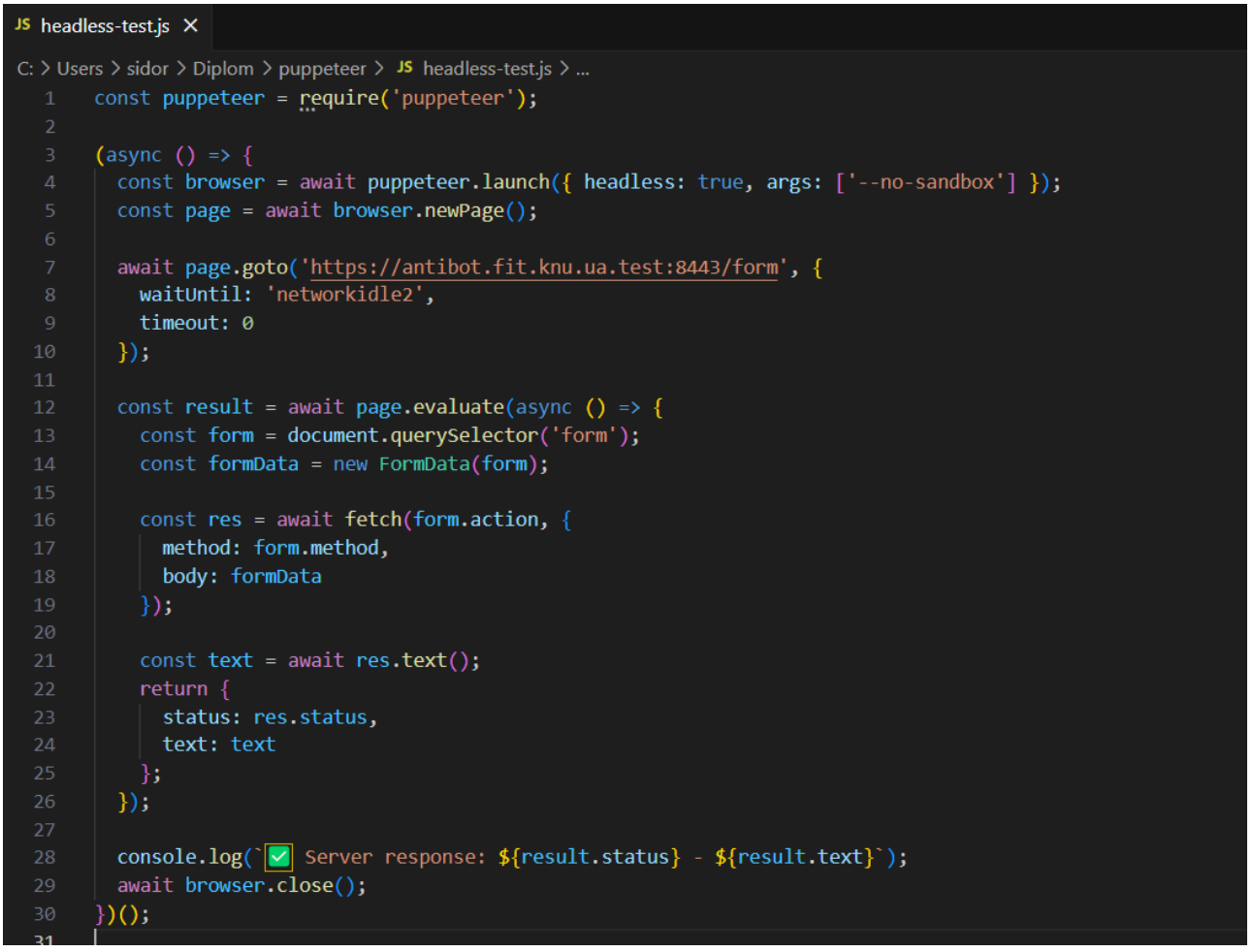


Рисунок 4.7 – Скрипт на основі Puppeteer

У результаті запуску скрипта консоль вивела повідомлення «Missing behavior check. Access denied» (рис. 4.8), а IP клієнта було заблоковано.

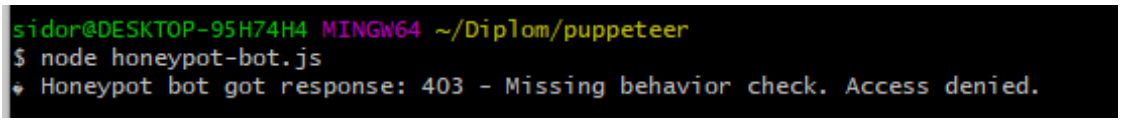


Рисунок 4.8 – Команда виконання скрипта без поведінкової перевірки

Це свідчить про правильне спрацювання логіки: бот не пройшов поведінкову перевірку, і сервер відмовив йому в доступі до форми. Такий механізм дозволяє виявляти headless-клієнтів, які ігнорують фронтенд-скрипти, навіть якщо інші параметри запиту виглядають коректно.

Ще одним важливим сценарієм тестування стала перевірка роботи honeypot-механізму, який реалізовано на рівні HTML-форми та контролера FormController. Ідея цього підходу полягає в додаванні до форми спеціального прихованого поля з назвою hidden_field, яке не видно користувачеві й не має бути заповненим при нормальній взаємодії через браузер. Якщо ж значення в цьому полі наявне, це майже гарантовано свідчить про роботу скрипта або бота, що бездумно заповнює всі поля форми.

Для моделювання такої ситуації був створений headless-бот на основі Puppeteer (рис. 4.9). Скрипт відкривав сторінку /form, знаходив усі поля input та автоматично заповнював їх. Honeypot-поле, будучи прихованим лише візуально, було також заповнене.

```javascript
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch({ headless: true, args: ['--no-sandbox'] });
  const page = await browser.newPage();
  await page.goto('https://antibot.fit.knu.ua.test:8443/form', {
    waitUntil: 'networkidle2',
    timeout: 0
  });
  const result = await page.evaluate(async () => {
    const inputs = document.querySelectorAll('input');
    inputs.forEach(input => {
      const type = input.getAttribute('type');
      const name = input.getAttribute('name');

      if (type === 'hidden' && name === 'form_created_at') {
        input.value = Date.now().toString();
      } else if (type === 'text') {
        input.value = 'bot';
      } else {
        input.value = 'test';
      }
    });
    const form = document.querySelector('form');
    const formData = new FormData(form);
    const res = await fetch(form.action, {
      method: form.method,
      body: formData
    });
    const text = await res.text();
    return {
      status: res.status,
      text: text
    };
  });
  console.log(`🤖 Honeypot bot got response: ${result.status} - ${result.text}`);
  await browser.close();
})();
```

Рисунок 4.9 – Скрипт на основі Puppeteer, який заповняє всі відскановані поля

Після заповнення даних скрипт сабмітив форму через fetch та отримав 403 помилку (рис. 4.10). У логах системи було зафіксовано, що поле «hidden_field» містило дані, і IP-адресу клієнта було негайно занесено до чорного списку за причиною «Honeypot field triggered». Повторне звернення до сервера з цього IP вже завершувалося помилкою 403 Forbidden.

```
sidor@DESKTOP-95H74H4 MINGW64 ~/Diplom/puppeteer
$ node honeypot-bot.js
🤖 Honeypot bot got response: 403 - Bot detected! Submission rejected.
```

Рисунок 4.10 – Команда виконання скрипта, який заповняє всі відскановані поля

Таким чином, honeypot-механізм успішно виявив автоматизований скрипт, який не розрізняв типи елементів у формі, що є типовою ознакою бот-активності. Це доводить ефективність even простих евристичних методів захисту, які не вимагають складної логіки або інтеграції зі сторонніми сервісами.

У слідуючому сценарії тестувалась ситуація, коли форма була відправлена надто швидко після її відкриття, що типово для ботів. Було використано Puppeteer-скрипт, який відкриває сторінку /form, одразу заповнює всі поля, включаючи обов'язкове «form_created_at», і миттєво надсилає форму (рис. 4.11).

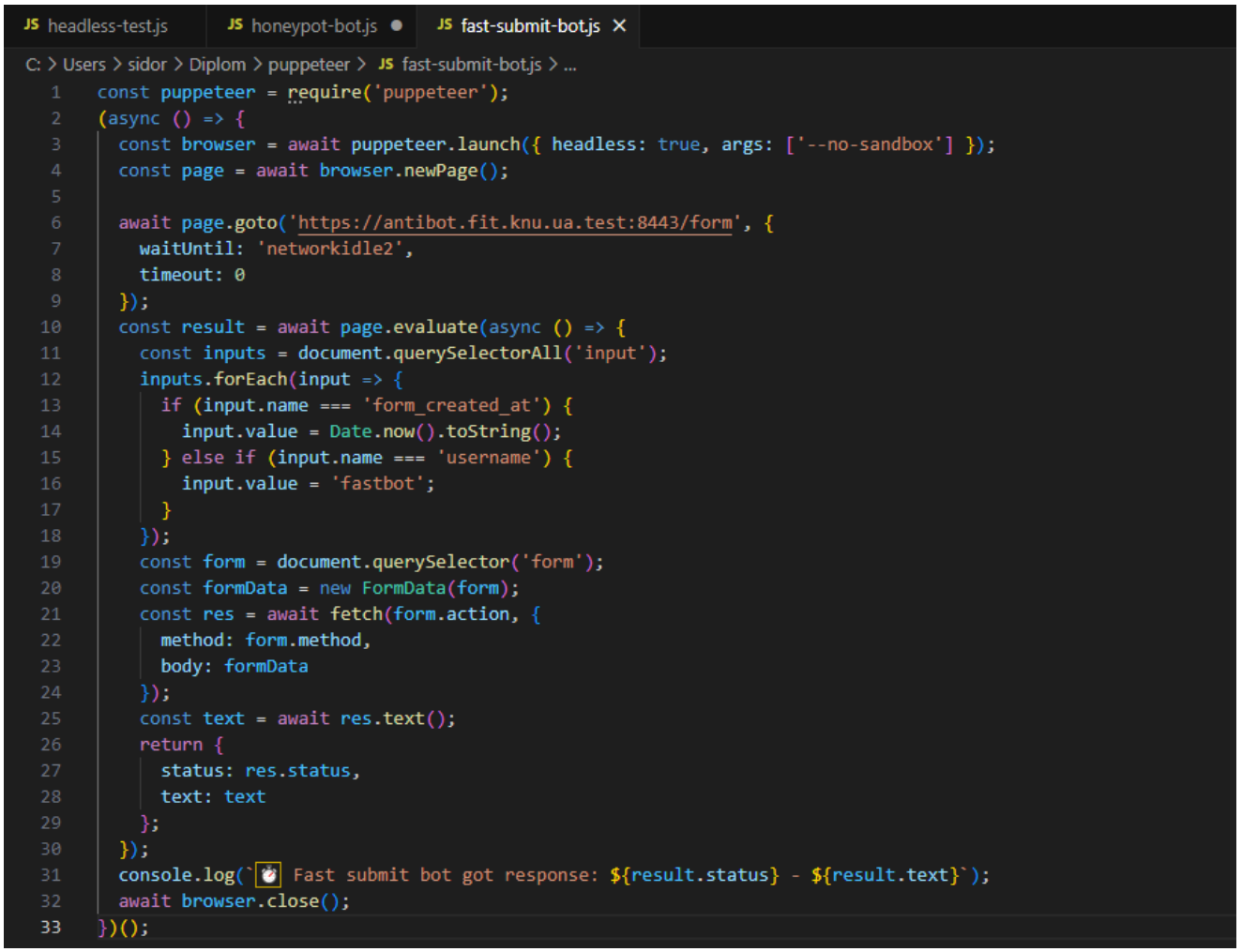


Рисунок 4.11 – Скрипт на основі Puppeteer, який миттєво відправляє форму

IP було заблоковано, а відповідь сервера вказує на те, що час заповнення був недостатнім для реальної взаємодії користувача (рис. 4.12).

```
sidor@DESKTOP-95H74H4 MINGW64 ~/Diplom/puppeteer
$ node fast-submit-bot.js
Fast submit bot got response: 403 - Form submitted too quickly. Access denied.
```

Рисунок 4.12 – Команда виконання скрипта, який миттєво відправляє форму

У цьому кейсі перевірялась поведінка системи в ситуації, коли з одного IP-адресу надсилається велика кількість запитів за короткий проміжок часу. На відміну від інших механізмів, що блокують IP негайно, у цьому випадку система застосовує адаптивний захист – перенаправляє користувача на сторінку з CAPTCHA для підтвердження, що він не є ботом.

Для моделювання цього сценарію був створений скрипт rate-limit-bot.js (рис. 4.13), який надсилає 12 послідовних GET-запитів до /form. Було навмисно обмежено автоматичне переспрямування (maxRedirects: 0), щоб у консолі можна було зафіксувати факт та напрямок редиректу.

JS headless-test.js | JS honeypot-bot.js | JS fast-submit-bot.js | JS rate-limit-bot.js

C: > Users > sidor > Diplom > puppeteer > JS rate-limit-bot.js > ...

```
1  const axios = require('axios');
2
3  (async () => {
4    for (let i = 0; i < 12; i++) {
5      try {
6        const res = await axios.get('https://antibot.fit.knu.ua.test:8443/form', {
7          headers: { 'User-Agent': 'Mozilla/5.0' },
8          maxRedirects: 0,
9          validateStatus: null,
10         httpsAgent: new (require('https').Agent)({ rejectUnauthorized: false })
11       });
12
13       const snippet = res.headers.location
14         ? `Redirect to ${res.headers.location}`
15         : res.data?.split('\n')[0]?.slice(0, 60) || 'No content';
16
17       console.log(`[${i + 1}] ${res.status} - ${snippet}`);
18     } catch (err) {
19       console.error(`[${i + 1}] ❌ ${err.message}`);
20     }
21   }
22 })();
```

Рисунок 4.13 – Скрипт на основі Puppeteer, який надсилає послідовно 12 запитів

У результаті перші 10 запитів були успішними (200 OK) і повертали звичайну HTML-сторінку форми. Починаючи з 11-го запиту, система відповіла статусом 302 Found і перенаправила користувача на сторінку з CAPTCHA (рис. 4.14).

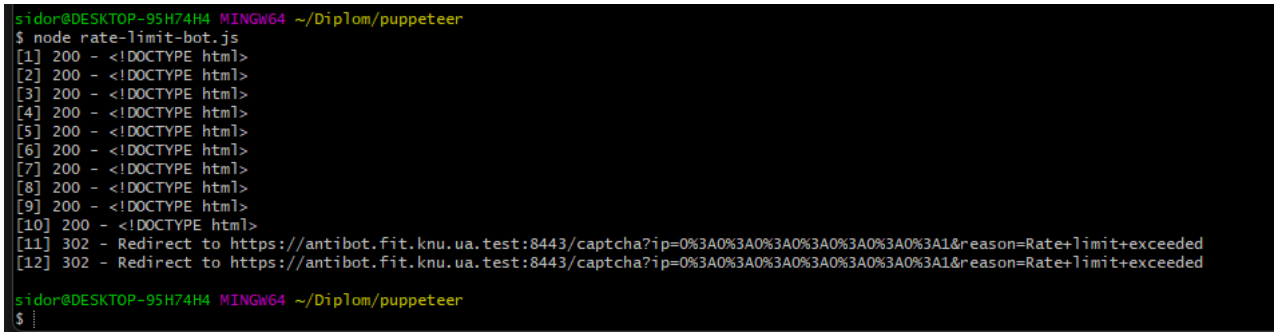


Рисунок 4.14 – Команда виконання скрипта, який надсилає послідовно 12 запитів

Це доводить, що система коректно фіксує перевищення частоти запитів і не допускає негайного доступу до ресурсу. Замість блокування IP, вона надає змогу пройти CAPTCHA – м’яка форма перевірки, яка дозволяє уникнути хибнопозитивних спрацювань, зберігаючи баланс між безпекою та зручністю.

У цьому кейсі перевіряється поведінка системи у випадку, коли IP-адреса була перенаправлена на CAPTCHA через перевищення ліміту запитів, але користувач проходить CAPTCHA-верифікацію успішно. Такий сценарій є типовим для легітимних користувачів, які надмірно активно взаємодіють із сервісом, однак не є ботами.

Для моделювання було відправлено запити до /form до моменту, поки сервер не перенаправив клієнта на сторінку /captcha. Потім на цій сторінці користувач заповнив CAPTCHA (використано Google reCAPTCHA v2, інтегровану у форму) та натиснув кнопку «Verify» (рис. 4.15).

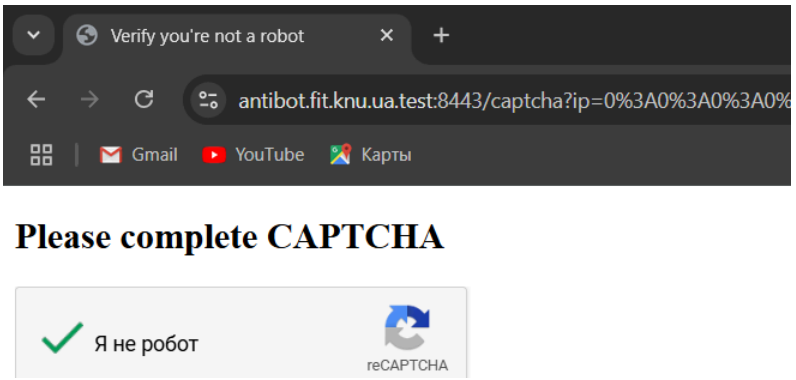


Рисунок 4.15 – Заповнена CAPTCHA

На бекенді цей запит обробляється маршрутом /captcha/verify. Контролер CaptchaController отримує токен reCAPTCHA разом із IP-адресою та причиною, і надсилає запит на API Google. Якщо валідація успішна, IP не блокується, і користувач перенаправляється назад до /form (рис. 4.16).

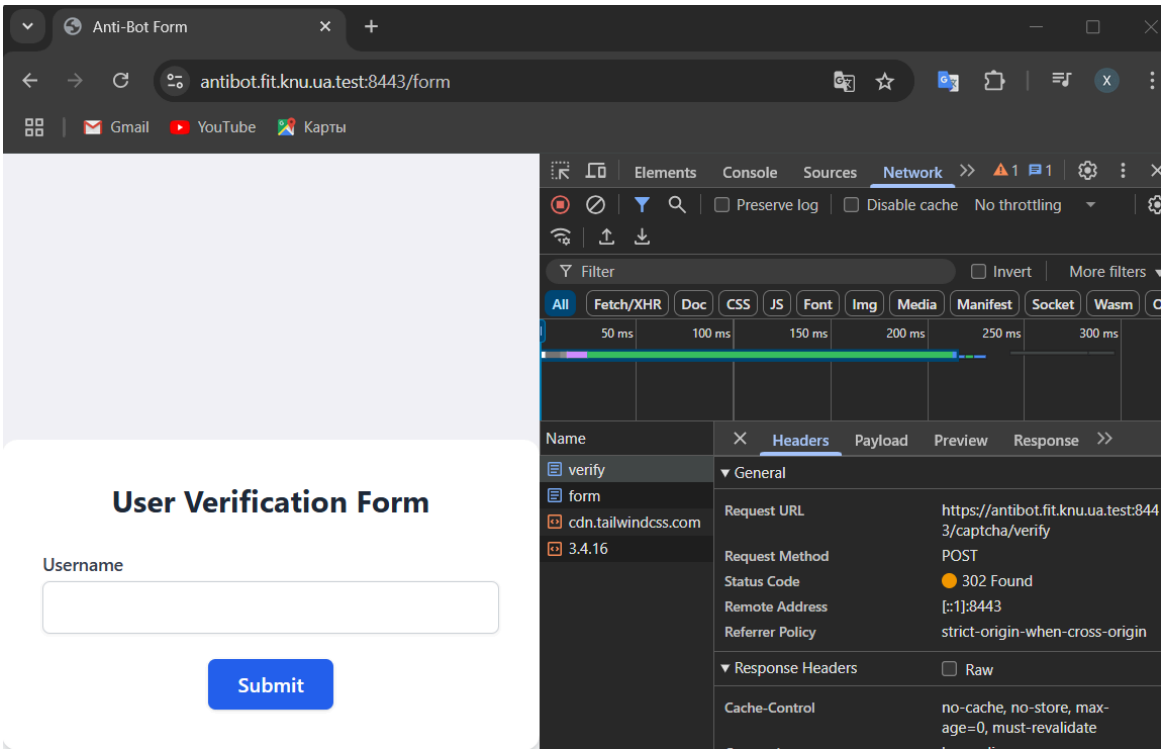


Рисунок 4.16 –Успішна валідація CAPTCHA

Це підтверджується тим, що після проходження CAPTCHA подальші запити з того самого IP знову отримують 200 OK, а база заблокованих IP BLACKLISTED_IPS не містить відповідного запису.

Таким чином, механізм CAPTCHA працює як м’який фільтр: замість негайного блокування користувача, йому пропонується пройти просту перевірку. Якщо він успішно її проходить – його IP не вноситься до чорного списку, і доступ до ресурсу залишається відкритим. Це дозволяє значно знизити кількість помилкових блокувань легітимних користувачів.

У останньому сценарії перевіряється позитивний варіант взаємодії з системою: легітимний користувач у браузері відкриває форму, взаємодіє з нею відповідно до очікуваної поведінки (рух миші, натискання клавіш), не заповнює приховані поля, не перевищує ліміти запитів і сабмітить форму з реалістичним таймінгом. Метою є перевірка, що в такому випадку система не заблокує IP, а дозволить виконати дію.

Користувач вручну перейшов на сторінку https://antibot.fit.knu.ua.test:8443/form у браузері Chrome, ввів дані у видиме поле username, після чого натиснув кнопку «Submit».

Результат видно в інтерфейсі розробника браузера (рис.4.17): запит до /submit мав статус 200 OK, а у відповіді сервер повернув повідомлення Hello, World! Submission accepted.

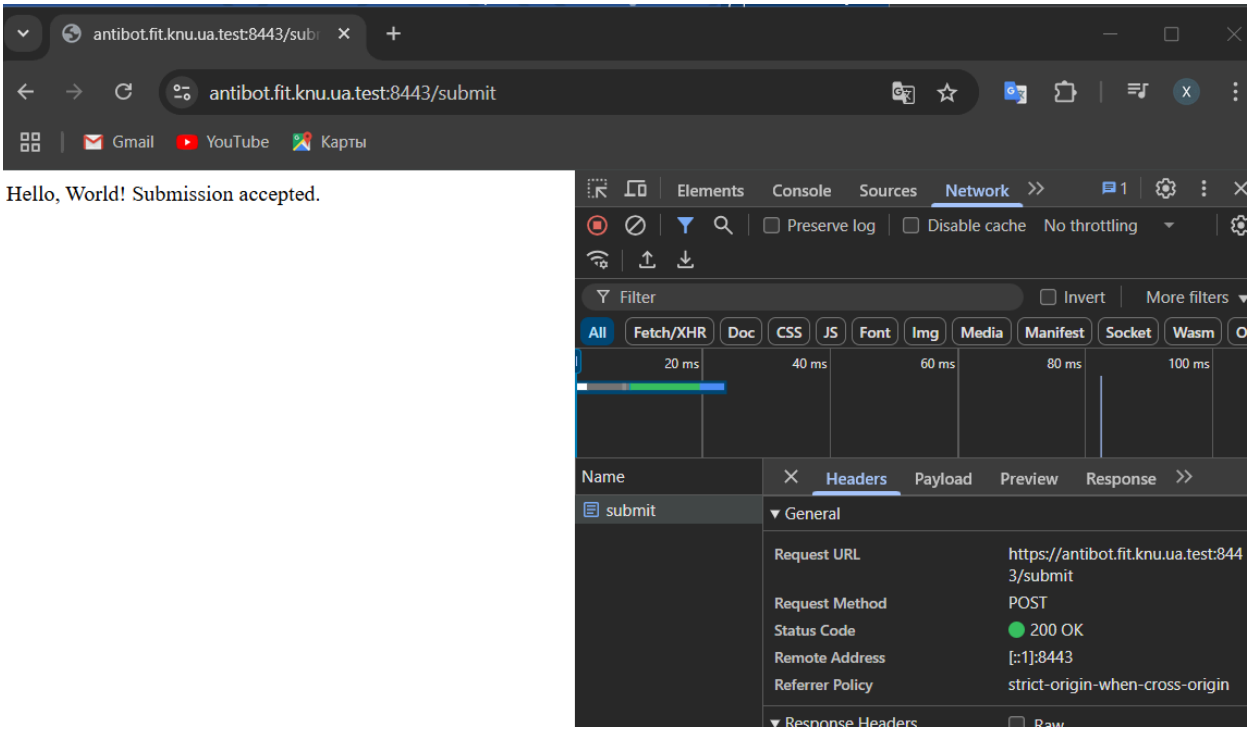


Рисунок 4.17 –Успішна відправка форми легітимним користувачем у браузері

Таким чином, система відпрацьовує не лише блокування ботів, а й забезпечує надійний доступ для легітимних клієнтів, що є критично важливою рисою ефективного захисту – мінімізація хибнопозитивних спрацювань.

4.2 Результати тестування

Для оцінки ефективності реалізованої системи захисту було проведено дев'ять окремих тестових сценаріїв, кожен з яких відповідав одному з механізмів виявлення ботів або перевірки легітимності користувача. Кожен сценарій мав чітко сформульовану мету: перевірити окрему захисну функцію, спричинити відповідну реакцію системи та зафіксувати результат у вигляді HTTP-відповіді, логів, записів у базі даних або поведінки інтерфейсу користувача. Загальна мета – підтвердити, що кожен із компонентів системи працює відповідно до очікуваної логіки: блокує підозрілу активність, перенаправляє на CAPTCHA у разі перевищення навантаження або пропускає легітимного користувача без перешкод. Результати тестування було зафіксовано в таблиці 4.1.

Таблиця 4.1

Результати тестування

№	Сценарій	Мета	Очікувана реакція	Результат
1	Відсутній User-Agent	Виявити запит без User-Agent	Блокування IP	Спрацювало
2	Некоректний Referer	Виявити підроблений Referer	Блокування IP	Спрацювало
3	Відсутній Origin	Виявити нестандартне джерело	Блокування IP	Спрацювало
4	Відсутність поведінкової взаємодії	Виявити бота без JS-активності	Блокування IP	Спрацювало

продовження таблиці 4.1

5	Honeypot-поле заповнене	Виявити автоматичне заповнення	Блокування IP	Спрацювало
6	Занадто швидкий сабміт	Виявити надшвидкий запит	Блокування IP	Спрацювало

7	Rate limit перевищено	Виявити надмірну активність	Перенаправлення на CAPTCHA	Спрацювало
8	Успішне проходження CAPTCHA	Довести, що користувач не бот	Дозвіл доступу	Спрацювало
9	Легітимна поведінка користувача	Перевірити відсутність блокувань	Дозвіл доступу	Спрацювало

У результаті тестування всі механізми спрацювали коректно відповідно до поставлених цілей. Із шести тестів, метою яких було саме блокування бота (сценарії 1–6), у всіх випадках IP-адреса була негайно заблокована, що становить 100% точність виконання цього захисту. У випадках сценаріїв 7 і 8, де система повинна була активувати адаптивний захист через CAPTCHA, механізм перенаправлення спрацював успішно, а також була перевірена можливість проходження CAPTCHA без блокування. Останній сценарій, у якому реальний користувач виконує всі дії через браузер, також завершився успішно без жодного втручання системи, що підтверджує її здатність не створювати бар’єрів для легітимних користувачів. Таким чином, усі 9 захисних сценаріїв виконали свою ціль, продемонструвавши коректність реалізації логіки та взаємодії між компонентами системи.

Висновки за розділом 4

У межах четвертого розділу було здійснено всебічне тестування впровадженого методу захисту Java веб-сервісу від шкідливих ботів. Основною метою цього етапу стала практична перевірка ефективності створеної системи, яка поєднує декілька рівнів контролю: перевірку заголовків HTTP-запитів, фільтрацію поведінкових аномалій, обмеження частоти звернень, блокування підозрілих IP-адрес і використання механізму CAPTCHA для додаткової валідації користувача. Комплексність реалізованої моделі дозволила охопити як класичні прояви бот-активності, так і сучасні сценарії, які характеризуються високим ступенем маскування імітованої поведінки під людську.

Для перевірки працездатності системи було змодельовано низку типових атак, зокрема: запити з порушеними або відсутніми заголовками User-Agent, Referer, Origin; headless-поведінка на основі бібліотек автоматизації; заповнення прихованих форм; надшвидке надсилання POST-запитів без затримок; надмірна частота звернень до одного ендпоінту; використання проксі та IP-спуфінгу для обходу обмежень. Усі ці варіанти охоплюють як базовий рівень загроз, так і високорівневі автоматизовані дії з використанням сучасних бот-фреймворків. Результати тестування підтвердили, що реалізований функціонал ефективно реагує на кожен із зазначених сценаріїв: спрацьовує перевірка на відсутність або фальсифікацію заголовків, коректно ідентифікуються headless-запити, блокується автоматизоване заповнення прихованих полів, спрацьовує обмеження на частоту запитів, а підозрілі дії перенаправляються на CAPTCHA.

Особливе значення має продемонстрована ефективність поведінкового фільтру. Завдяки збору та аналізу характеристик запиту, таких як затримки між діями, шаблони навігації, швидкість переходів і кількість звернень за одиницю часу, система змогла виявити аномалії у поведінці ботів навіть у випадках, коли заголовки були правдоподібними. Це свідчить про здатність методу виявляти складні сценарії, які маскують свою присутність через зовнішню схожість із легітимним трафіком. Водночас важливо відзначити, що всі тестові взаємодії справжніх користувачів пройшли без блокувань або суттєвих затримок, що підтверджує коректність визначення порогів реагування системи.

Суттєву роль у зменшенні хибнопозитивних блокувань відіграє застосування CAPTCHA як м’якої адаптивної реакції. Якщо IP-адреса не проходить базову перевірку, вона не блокується миттєво, а направляється на CAPTCHA, де користувач має змогу довести свою справжність. У випадку успішного проходження перевірки, доступ відновлюється, і активність користувача не маркується як ворожа. Це забезпечує баланс між рівнем безпеки і користувацькою зручністю, дозволяючи уникнути непотрібних відмов у доступі.

З технічного боку реалізація була виконана на базі Java та Spring Boot. Було створено набір фільтрів і сервісів, які реалізують усі перевірки, а також систему

збереження і обробки історії звернень до сервісу. Логи подій, що накопичуються, дозволяють проводити подальший аналіз активності, оптимізувати порогові значення перевірок і виявляти нові шаблони шкідливої поведінки.

Тестування під навантаженням також підтвердило стабільність роботи системи. Під час симульованих масових атак не було зафіксовано жодних збоїв, затримок у відповіді чи порушень доступності для легітимного користувача. Система встигала обробляти всі вхідні запити, виявляти підозрілі патерни, оновлювати статистику в реальному часі і застосовувати відповідні дії без необхідності зовнішнього втручання.

У підсумку можна стверджувати, що проведене тестування повністю підтвердило працездатність і практичну ефективність розробленого методу захисту. Його ключовими перевагами є багаторівневий підхід до аналізу запитів, можливість адаптації до змін у характері трафіку, здатність реагувати без порушення легітимного обслуговування, а також відкритість до розширення. У разі потреби, функціонал системи може бути доповнений машинним навчанням або інтегрований із зовнішніми платформами захисту. Таким чином, запропоноване рішення демонструє не лише ефективність у виявленні шкідливих ботів, а й відповідає вимогам масштабованості, надійності та реального впровадження в сучасні веб-інфраструктури.

ВИСНОВКИ

У ході виконання роботи було комплексно досліджено проблему захисту веб-сервісів від шкідливих ботів та розроблено ефективне програмне рішення для виявлення та нейтралізації таких загроз. Теоретична частина дослідження охопила класифікацію ботів, аналіз сучасних типів атак, розгляд методів детектування та систем реагування. Було виявлено, що шкідливі боти здатні здійснювати автоматизований збір даних, генерацію навантаження, зловживання API, а також формувати складні мережі ботнетів, які становлять суттєву загрозу для стабільної роботи онлайн-сервісів.

На основі аналізу наявних підходів було сформовано архітектуру багаторівневої системи захисту, яка поєднує як евристичні, так і поведінкові механізми виявлення ботів, а також адаптивну перевірку за допомогою CAPTCHA. Для реалізації рішення було використано сучасні технології Java-екосистеми: Spring Boot, Spring Security, Spring Data JPA, а також Caffeine, Flyway та інтеграцію з Google reCAPTCHA. Архітектура побудована з урахуванням принципів чистої архітектури, модульності та розширюваності.

У практичній частині дипломної роботи реалізовано такі механізми: перевірка HTTP-заголовків (User-Agent, Referer, Origin), фіксація поведінки користувача через JavaScript, honeypot-поля, аналіз часу заповнення форми, обмеження частоти запитів (RateLimiterService) та адаптивне перенаправлення на CAPTCHA. Було впроваджено централізовану систему блокування IP-адрес з фіксацією причин, часу блокування та можливістю контролю.

Проведене тестування показало, що всі реалізовані механізми працюють узгоджено та ефективно. Усі моделювання типових бот-сценаріїв завершилися відповідним блокуванням або захисною дією, а справжні користувачі змогли безперешкодно взаємодіяти із сервісом. Система довела свою здатність до гнучкого реагування, точного розмежування трафіку та збереження доступності для легітимних клієнтів.

У підсумку, розроблене рішення є практично орієнтованою моделлю захисту веб-додатків від автоматизованої активності. Воно може бути масштабоване та адаптоване для інтеграції в реальні веб-сервіси різного рівня складності, що підтверджує його прикладну цінність та актуальність у сучасному цифровому середовищі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IBM What is a REST API? [Електронний ресурс]. – Режим доступу: https://www.ibm.com/topics/rest-apis

2. OWASP API Security Project [Електронний ресурс]. – Режим доступу: https://owasp.org/www-project-api-security/

3. Madden N. API Security in Action. – 2020. – 576 p.

4. Yevdokymov O. Spring Framework in Java – why is it worth your attention? [Електронний ресурс]. – Режим доступу: https://svitla.com/blog/spring-framework-in-java-why-is-it-worth-your-attention

5. Мирутенко Л.В., Сидоренко Б.М. Методи виявлення та запобіганням шкідливим ботам у Java веб-сервісах // VIII Міжнародна науково-практична конференція «Проблеми кібербезпеки інформаційно-комунікаційних систем» (PCSICS): зб. матеріалів. 2025. С. 142-143

6. Spring Framework Documentation [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-framework/reference/

7. Spring Security Reference Documentation [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-security/reference/

8. Spring Data JPA Documentation [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-data/jpa/

9. Flyway Documentation [Електронний ресурс]. – Режим доступу: https://flywaydb.org/documentation

10. Google Developers. reCAPTCHA Documentation [Електронний ресурс]. – Режим доступу: https://developers.google.com/recaptcha

11. Caffeine Cache Documentation [Електронний ресурс]. – Режим доступу: https://github.com/ben-manes/caffeine

12. Imperva. Bad Bots in 2025: The Growing Business Risk You Can’t Ignore [Електронний ресурс]. – Режим доступу:

https://www.imperva.com/resources/resource-library/webinars/bad-bots-in-2025-the-growing-business-risk-you-cant-ignore/

13. Cloudflare. Bot Management [Електронний ресурс]. – Режим доступу: https://www.cloudflare.com/products/bot-management/

14. Imperva. The Economic Impact of API and Bot Attacks. – 2024 [Електронний ресурс]. – Режим доступу: https://www.imperva.com/resources/resource-library/reports/the-economic-impact-of-api-and-bot-attacks/

15. Spring Boot Documentation [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-boot/documentation.html

16. Mozilla. Web Security Guidelines [Електронний ресурс]. – Режим доступу: https://infosec.mozilla.org/guidelines/web_security

17. Selenium Project. Documentation [Електронний ресурс]. – Режим доступу: https://www.selenium.dev/documentation/

18. Intoli. Detecting Chrome Headless [Електронний ресурс]. – Режим доступу: https://intoli.com/blog/not-possible-to-block-chrome-headless/

19. Google. How Google prevents invalid traffic [Електронний ресурс]. – Режим доступу: https://www.google.com/ads/adtrafficquality/how-we-prevent-it/

20. DataDome. How effective is CAPTCHA? Why it’s not enough for bot protection… [Електронний ресурс]. – Режим доступу: https://datadome.co/guides/captcha/traditional-captcha-obsolete/

21. Baeldung. Spring Boot and Caffeine Cache [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/spring-boot-caffeine-cache

22. Microsoft. WAF on Azure Application Gateway bot protection overview [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/azure/web-application-firewall/ag/bot-protection-overview

23. OWASP Foundation. OWASP Top 10:2021 [Електронний ресурс]. – Режим доступу: https://owasp.org/Top10/

24. Google Cloud. Application Security Best Practices [Електронний ресурс]. – Режим доступу: https://cloud.google.com/security/best-practices

25. Oracle. Secure Coding Guidelines for Java SE [Електронний ресурс]. – Режим доступу: https://www.oracle.com/java/technologies/javase/seccodeguide.html

26. Meng N., Nagy S., Yao D., Zhuang W., Arango Argoty G. Secure Coding Practices in Java: Challenges and Vulnerabilities [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/1709.09970

27. Escape. 10 Best Practices to Secure Your Spring Boot Applications [Електронний ресурс]. – Режим доступу: https://escape.tech/blog/security-best-practices-for-spring-boot-applications/

28. Spring. Securing a Web Application [Електронний ресурс]. – Режим доступу: https://spring.io/guides/gs/securing-web/

29. VM Software House. 12 Security Best Practices in Java Development [Електронний ресурс]. – Режим доступу: https://vmsoftwarehouse.com/12-security-best-practices-in-java-development

30. University of California, Berkeley. Java Security Best Practices [Електронний ресурс]. – Режим доступу: https://security.berkeley.edu/education-awareness/java-security-best-practices

31. Java.com. Tips for Using Java Securely [Електронний ресурс]. – Режим доступу: https://www.java.com/en/download/help/security-tips.html

32. DataDome. Bot Detection – How to Detect Bots on Your Website, Apps, & APIs [Електронний ресурс]. – Режим доступу: https://datadome.co/guides/bot-protection/bot-detection-how-to-identify-bot-traffic-to-your-website/

33. Radware. Bot Detection [Електронний ресурс]. – Режим доступу: https://www.radware.com/cyberpedia/bot-management/bot-detection/

34. ZenRows. How to Avoid Bot Detection With Selenium [Електронний ресурс]. – Режим доступу: https://www.zenrows.com/blog/selenium-avoid-bot-detection

35. ZenRows. Bypass Bot Detection (2025): 5 Best Methods [Електронний ресурс]. – Режим доступу: https://www.zenrows.com/blog/bypass-bot-detection

36. Captcha.eu. What are Advanced Persistent Bots (APB)? [Електронний ресурс]. – Режим доступу: https://www.captcha.eu/uk/what-are-advanced-persistent-bots-apb/

37. OWASP Foundation. HTTP Headers Cheat Sheet [Електронний ресурс]. – Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html

38. Mozilla Developer Network. Introduction to HTTP Headers [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

39. OWASP Foundation. Identify Application Entry Points – Web Security Testing Guide [Електронний ресурс]. – Режим доступу: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/01-Information_Gathering/06-Identify_Application_Entry_Points

40. Cloudflare. JavaScript Detections – Cloudflare Challenges Documentation [Електронний ресурс]. – Режим доступу: https://developers.cloudflare.com/cloudflare-challenges/challenge-types/javascript-detections/

41.

ДОДАТОК А

Блок-схема розробленого методу захисту від шкідливих ботів в Java веб-сервісах

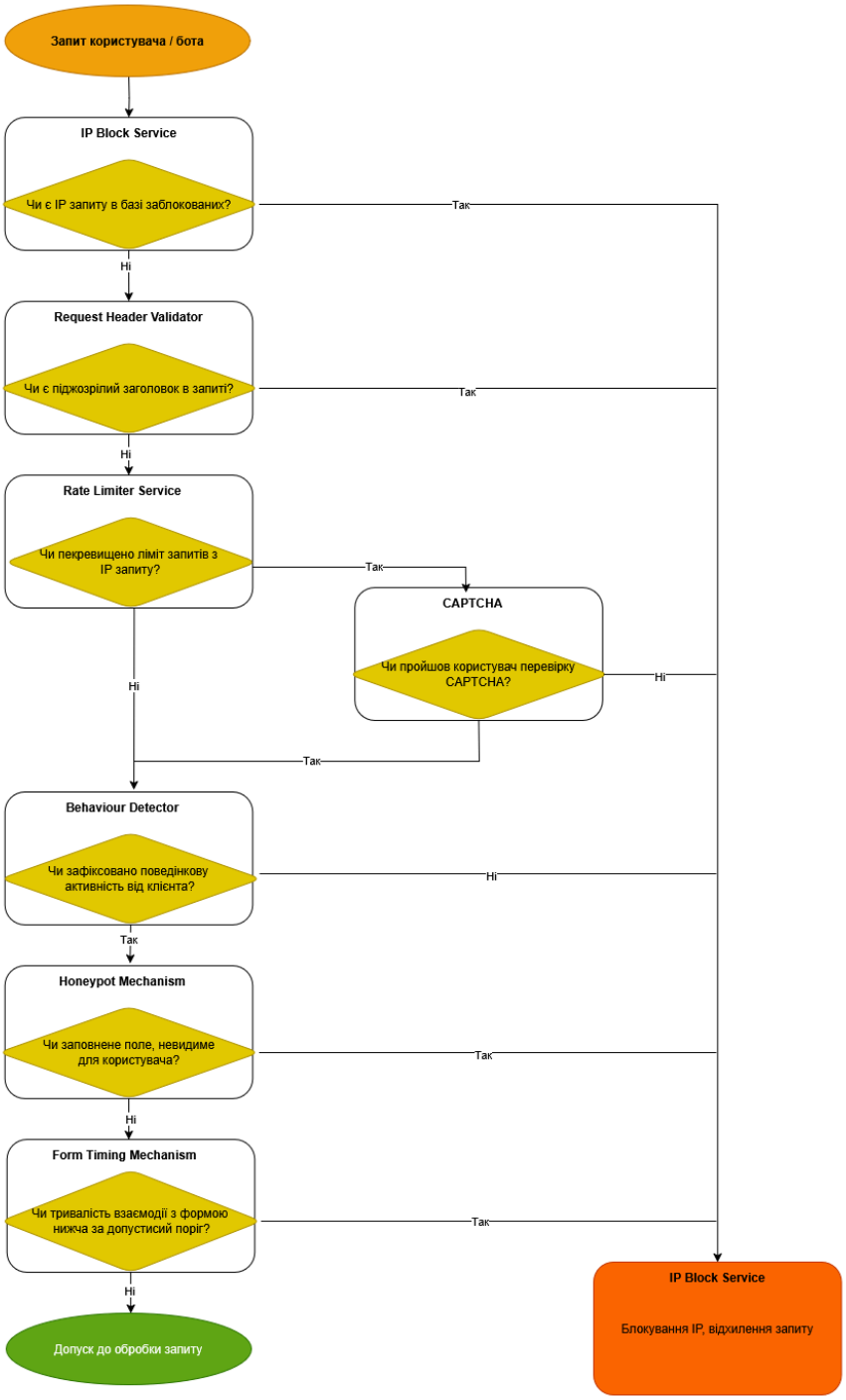


Рисунок А.1 – Блок-схема розробленого методу захисту від шкідливих ботів у Java веб-сервісі

ДОДАТОК Б

Файл form.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <title>Anti-Bot Form</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <script>
    window.addEventListener('DOMContentLoaded', () => {
      document.getElementById('form_created_at').value = Date.now();
    });
  </script>
</head>
<body class="bg-gray-100 flex items-center justify-center min-h-screen">
<div class="bg-white shadow-md rounded-xl p-8 max-w-md w-full">
    <h2 class="text-2xl font-bold text-gray-800 mb-6 text-center">User Verification Form</h2>

    <form id="userForm" action="/submit" method="post" class="space-y-5">
        <!-- Error Box -->
        <div id="errorBox" class="hidden bg-red-100 text-red-700 border border-red-300 px-4 py-3 rounded-md text-sm"></div>

        <!-- Username Field -->
        <div>
```

```
        <label for="username" class="block text-sm font-medium
text-gray-700">Username</label>
        <input type="text" id="username" name="username" required
            class="mt-1 w-full px-4 py-2 border rounded-md shadow-sm focus:ring
focus:ring-blue-200 focus:outline-none border-gray-300" />
    </div>

    <!-- Honeypot Field -->
    <input type="text" name="hidden_field" style="display:none;" tabindex="-1"
autocomplete="off" />

    <!-- Timestamp Field -->
    <input type="hidden" id="form_created_at" name="form_created_at" />

    <!-- Submit Button -->
    <div class="text-center">
      <button type="submit"
          class="bg-blue-600 hover:bg-blue-700 text-white font-semibold px-6 py-2
rounded-md shadow">
        Submit
      </button>
    </div>
  </form>
</div>

<script>
  const behavior = {
    hasMouseMove: false,
    hasKeyPress: false,
    headless: navigator.webdriver
```

```
};

window.addEventListener("mousemove", () => behavior.hasMouseMove = true);
window.addEventListener("keydown", () => behavior.hasKeyPress = true);

document.getElementById('userForm').addEventListener('submit', async function (e) {
  e.preventDefault();
  const form = e.target;

  const behaviorResponse = await fetch('/behavior', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(behavior)
  });

  if (behaviorResponse.status === 403) {
    const text = await behaviorResponse.text();
    const errorBox = document.getElementById("errorBox");
    errorBox.classList.remove("hidden");
    errorBox.innerText = text;
    return;
  }

  const formData = new FormData(form);
  const res = await fetch(form.action, {
    method: 'POST',
    body: formData
```

```
    });

    if (res.status === 403) {
      const text = await res.text();
      const errorBox = document.getElementById("errorBox");
      errorBox.classList.remove("hidden");
      errorBox.innerText = text;
    } else {
      form.removeEventListener('submit', arguments.callee);
      form.submit();
    }
  });
</script>
</body>
</html>
```

ДОДАТОК В

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Тези наукових доповідей

1. Мирутенко Л.В., Сидоренко Б.М. Методи виявлення та запобіганням шкідливим ботам у Java веб-сервісах. VIII Міжнародна науково-практична конференція «Проблеми кібербезпеки інформаційно-комунікаційних систем» (PCSICS). 2025. С. 142-143