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

«__» червня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)
спеціальність 125 Кібербезпека
(код і назва спеціальності)
освітній ступень бакалавр
освітня програма Кібербезпека
(назва освітньо-професійної програми)
на тему: «Механізми безпеки вебдодатків на основі мови програмування Go (Golang)»

Виконавець: студент IV курсу, групи КБ-41

_____ **Ілля ІЗМАЛКОВ**
(підпис) (ім'я, прізвище)

	Підпис	**Ім'я, прізвище**
Керівник		Сергій БУЧИК
Нормоконтроль		Леся БАРАНОВСЬКА

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

29 листопада 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студенту **КБ-41** **Ізмалкову Іллі Романовичу**
(група) (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Механізми безпеки вебдодатків на основі мови програмування Go (Golang)

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №6 від 28.11.2024 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Засоби та методи забезпечення безпеки вебдодатків

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Необхідно ознайомитись із сучасним станом безпеки вебдодатків, виявити основні загрози та проаналізувати особливості застосування мови Go для розробки безпечних вебдодатків

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Застосування розроблених матеріалів для підвищення захищеності вебдодатків на основі мови програмування Go.

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 29 листопада 2024 року

Завдання видав	_____ (підпис)	Сергій БУЧИК (ім'я, прізвище)
Завдання прийняв до виконання	_____ (підпис)	Ілля ІЗМАЛКОВ (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	29.11.2024 – 05.12.2024	*виконано*
2	Аналіз літератури	06.12.2024 – 29.12.2024	*виконано*
3	Обґрунтування вибору методів дослідження	15.01.2025 – 29.01.2025	*виконано*
4	Аналіз мови програмування Go для розробки безпечних вебдодатків	30.01.2025 – 14.02.2025	*виконано*
5	Дослідження типових вразливостей згідно з OWASP Top 10 та OWASP Go Secure Coding Practices	15.02.2025 – 28.02.2025	*виконано*
6	Розробка демонстраційного вебдодатку з урахуванням вимог безпеки, тестування	29.02.2025 – 12.03.2025	*виконано*
7	Формування практичних рекомендацій та написання висновків	13.03.2025 – 29.03.2025	*виконано*
8	Написання тексту атестаційної роботи	01.04.2025 – 15.05.2025	*виконано*
9	Оформлення пояснювальної записки	16.05.2025 – 23.05.2025	*виконано*
10	Підготовка до захисту кваліфікаційної роботи	24.05.2025 – 13.06.2025	*виконано*

Завдання видав	_____ (підпис)	Сергій БУЧИК (ім'я, прізвище)
Завдання прийняв до виконання	_____ (підпис)	Ілля ІЗМАЛКОВ (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 13 червня 2025 року

РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. 70 стор., 2 табл., 4 рис., 31 лістинг., 25 джер., 2 додатки.

Метою роботи є розробка практичних підходів до підвищення безпеки вебдодатків на основі мови програмування Go з урахуванням сучасних типових загроз та рекомендацій OWASP.

Об'єктом дослідження є процес забезпечення кібербезпеки вебдодатків на основі мови програмування Go.

Предметом дослідження є методи виявлення, аналізу та запобігання типовим вразливостям вебдодатків на основі безпечного програмування мовою Go відповідно до OWASP Top 10 та Go Secure Coding Practices.

Під час виконання кваліфікаційної роботи було використано такі методи дослідження:

– аналіз наукової літератури та відкритих джерел;

– порівняння;

– синтез;

– експеримент.

Практична цінність отриманих результатів полягає в можливому впровадженні розроблених підходів для підвищення рівня захищеності вебдодатків на основі мови програмування Go.

Ключові слова: вебдодаток, мова програмування Go, OWASP, вразливості, ZAP

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

CSP	—	Communicating Sequential Processes
I/O	—	Ввід/Вивід (Input/Output)
GCC	—	GNU Compiler Collection
GNU	—	GNU is Not Unix
API	—	Application Programming Interface
ОС	—	Операційна система
OWASP	—	Open Worldwide Application Security Project
URL	—	Uniform Resource Locator
SQL	—	Structured Query Language
XSS	—	Cross-Site Scripting
БД	—	База даних
СУБД	—	Система управління базами даних
HTTP	—	HyperText Transfer Protocol
HTTPS	—	HyperText Transfer Protocol Secure
CSP	—	Content Security Policy
ZAP	—	Zed Attack Proxy by Checkmarx
ППЗ	—	Проміжне програмне забезпечення
CSRF	—	Cross-Site Request Forgery
HSTS	—	HTTP Strict Transport Security

ВСТУП

У сучасному цифровому середовищі вебдодатки стали ключовим елементом інформаційної інфраструктури як у приватному, так і в державному секторах. Щоденно мільйони користувачів взаємодіють із веб-сервісами, передаючи конфіденційні дані, здійснюючи фінансові операції та використовуючи ресурси, що зберігають важливу інформацію. З огляду на це, питання безпеки вебдодатків набуває критичного значення.

За умови постійного зростання кількості кіберзагроз та підвищення складності атак на вебдодатки вибір мови програмування з вбудованими засобами безпеки та широкою екосистемою стає критично важливим. Однією з таких мов, що стрімко набирає популярність, є Go (Golang) — проект з відкритим початковим кодом, розроблений в Google. Вона поєднує високу продуктивність та зручність у використанні із стійкістю до поширених вразливостей завдяки суворій системі типів, безпеці пам'яті та потужній стандартній бібліотеці. Go успішно використовується масштабними інфраструктурними проєктами (Docker, Kubernetes, Traefik) і великими корпораціями (ByteDance, Google, Uber тощо), що свідчить про її надійність та здатність витримувати високі навантаження й забезпечувати захищеність сервісів. Зростання популярності та активне впровадження у критично важливі продукти зумовлює своєчасність та практичну важливість дослідження методів захисту вебдодатків на базі цієї мови. Таким чином, дослідження можливостей та інструментів забезпечення безпеки у вебдодатках, створених на Go, є не лише науково цікавою, але й практично важливою задачею, що сприятиме підвищенню рівня кібербезпеки.

Метою роботи є розробка практичних підходів до підвищення безпеки вебдодатків на основі мови програмування Go з урахуванням сучасних типових загроз та рекомендацій OWASP.

Для досягнення поставленої мети було визначено наступні завдання:

– проаналізувати особливості мови програмування Go в розрізі безпечної розробки вебдодатків;

– дослідити типові загрози безпеці вебдодатків згідно з OWASP Top 10;

– проаналізувати рекомендації OWASP Go Secure Coding Practices та їх відповідність до загроз OWASP;

– реалізувати приклад безпечного кодування вебдодатків на основі мови програмування Go з використанням інструментів для автоматизованого виявлення вразливостей.

Об'єктом дослідження є процес забезпечення кібербезпеки вебдодатків на основі мови програмування Go.

Предметом дослідження є методи виявлення, аналізу та запобігання типовим вразливостям вебдодатків на основі безпечного програмування мовою Go відповідно до OWASP Top 10 та Go Secure Coding Practices.

Під час виконання кваліфікаційної роботи було використано такі методи дослідження:

– аналіз наукової літератури та відкритих джерел;

– порівняння;

– синтез;

– експеримент.

Практична цінність отриманих результатів полягає в можливому впровадженні розроблених підходів для підвищення рівня захищеності вебдодатків на основі мови програмування Go.

Розділ 1. ОГЛЯД МОВИ ПРОГРАМУВАННЯ GO

Go — компільована рівночасна* мова програмування зі статичною типізацією та збиранням сміття, що розробляється в Google. Проєкт має відкритий початковий код. Її було задумано як спробу вирішення певних проблем, з якими стикалися розробники програмної інфраструктури в Google. Це — проблеми, пов'язані з багатоядерними процесорами, мережевими системами, масивними обчислювальними кластерами та моделлю веб-програмування в цілому [1]. Станом на травень 2025 року мова займає сьому сходинку в рейтингу популярності за індексом TIOBE [2]. В цьому розділі розглянуто історію створення та принципи дизайну, основні концепції та особливості Go, зокрема модель рівночасності в порівнянні з іншими мовами програмування.

1.1 Історія створення

1.1.1 Передумови виникнення

На момент зародження мови Go** у вересні 2007 року в Google вже відчули, що існуючі мови — передусім C++ і Java — стали надто громіздкими для швидкої розробки великих розподілених систем; при цьому апаратні платформи лише починали переходити на багатоядерність, але мови програмування майже не допомагали користуватися цим ресурсом ефективно. Ще більше ускладнювали життя складні системи збірки й відсутність єдиних інструментів, що призводило до тривалого часу компіляції, небезпечного ручного управління пам'яттю й проблем із масштабуванням великих кодових баз [4].

Ось більш детальний перелік проблем під час розробки великих проєктів в Google, на які зважали розробники Go [1]:

---

*Переклад терміну «concurrent» (Англійсько-українсько-англійський словник наукової мови (Фізика та споріднені науки) у двох частинах).

**Варто зазначити, що назва мови програмування — саме Go, а не Golang. Останнє походить від доменного імені golang.org [3].

– Повільна компіляція: Час компіляції великих проектів міг сягати хвилин або навіть годин, що уповільнювало цикл розробки та тестування.

– Неконтрольовані залежності: У великих проектах важко було відстежувати та контролювати залежності між модулями, що призводило до складнощів у підтримці та масштабуванні систем.

– Кожен програміст використовує різний підмножину мови: Різні стилі кодування та використання різних можливостей мови ускладнювали спільну роботу над проектами.

– Слабке розуміння програми: Код був важким для читання, погано документованим, що ускладнювало його підтримку та розвиток.

– Дублювання зусиль: Відсутність централізованих рішень призводила до повторного вирішення одних і тих же проблем різними командами.

– Вартість оновлень: Оновлення залежностей або компонентів вимагали значних зусиль і часу.

– Розбіжності версій: Використання різних версій одних і тих же бібліотек призводило до конфліктів та помилок.

– Складність написання автоматичних інструментів: Складність мови та її особливості ускладнювали створення інструментів для автоматизації процесів розробки.

– Крос-мовні збірки: Інтеграція компонентів, написаних на різних мовах, ускладнювала процес складання та тестування систем.

Метою проекту було усунути ці недоліки та зробити процес розробки більш продуктивним і масштабованим, тому дизайн Go спрямований саме на підвищення продуктивності розробників, спрощення управління великими кодовими базами та ефективне використання сучасного апаратного забезпечення.

1.1.2 Принципи дизайну

Одним із ключових факторів, що мотивували створення Go, була непродуктивність та надлишкова складність існуючих мов програмування на великих

проєктах. Як зазначив Роб Пайк, один з авторів Go, «складність є мультиплікативною»: спроба вирішити одну проблему за допомогою складнішого рішення часто створює нові проблеми в інших частинах системи. Саме тому дизайн мови програмування Go ґрунтується на принципі *радикальної простоти* та прагненні до *концептуальної цілісності*, що передбачає жорстке дотримання спрощення на всіх рівнях: мови, інструментів, стандартних бібліотек і навіть культури розробки [5].

Go спроєктовано як мову з невеликою кількістю потужних засобів, які поєднуються в простий та передбачуваний спосіб. Зокрема:

– Сувора відмова від зайвого функціоналу: у мові відсутні успадкування, генератори, конструктори, деструктори, оператори перевантаження, винятки, макроси, анотації функцій тощо.

– Простота типів: Go має досить сильну, але мінімалістичну систему типів, яка дозволяє уникати багатьох помилок, притаманних динамічним мовам, але без зайвого формалізму, властивого мовам на кшталт Haskell чи C++.

– Стабільність і зворотна сумісність: код, написаний раніше, гарантовано компілюється на нових версіях компілятора та стандартної бібліотеки.

Ще одним важливим принципом Go є ефективна робота з пам’яттю і врахування особливостей сучасних апаратних архітектур. Типи, зокрема структури й масиви, зберігають дані безпосередньо, без додаткової індексації чи динамічних алокацій, що зменшує накладні витрати. Крім того, паралелізм і рівночасність є вбудованими концепціями мови завдяки механізмам горутин і каналів (на основі моделі CSP), що дозволяє легко писати масштабовані паралельні програми. Більш детальна розповідь про модель рівночасності Go знаходиться в підрозділі 1.6.

Go також робить акцент на уніфікованій системі інструментів і бібліотек. Стандартна бібліотека Go включає все необхідне для роботи з текстом, мережею, криптографією, графікою, I/O, а також для створення розподілених систем. Інструмент `go` дозволяє автоматизувати збирання, тестування, форматування й

документування коду без складних конфігурацій — лише за допомогою структури проєкту та конвенцій.

Таким чином, принципи дизайну Go полягають у:

– мінімалізмі функцій без втрати можливостей;

– простоті синтаксису та архітектури коду;

– ефективній роботі з ресурсами;

– забезпеченні стабільності та сумісності;

– максимальній автоматизації за допомогою конвенцій, а не конфігурацій.

1.1.3 Мови, що вплинули на Go

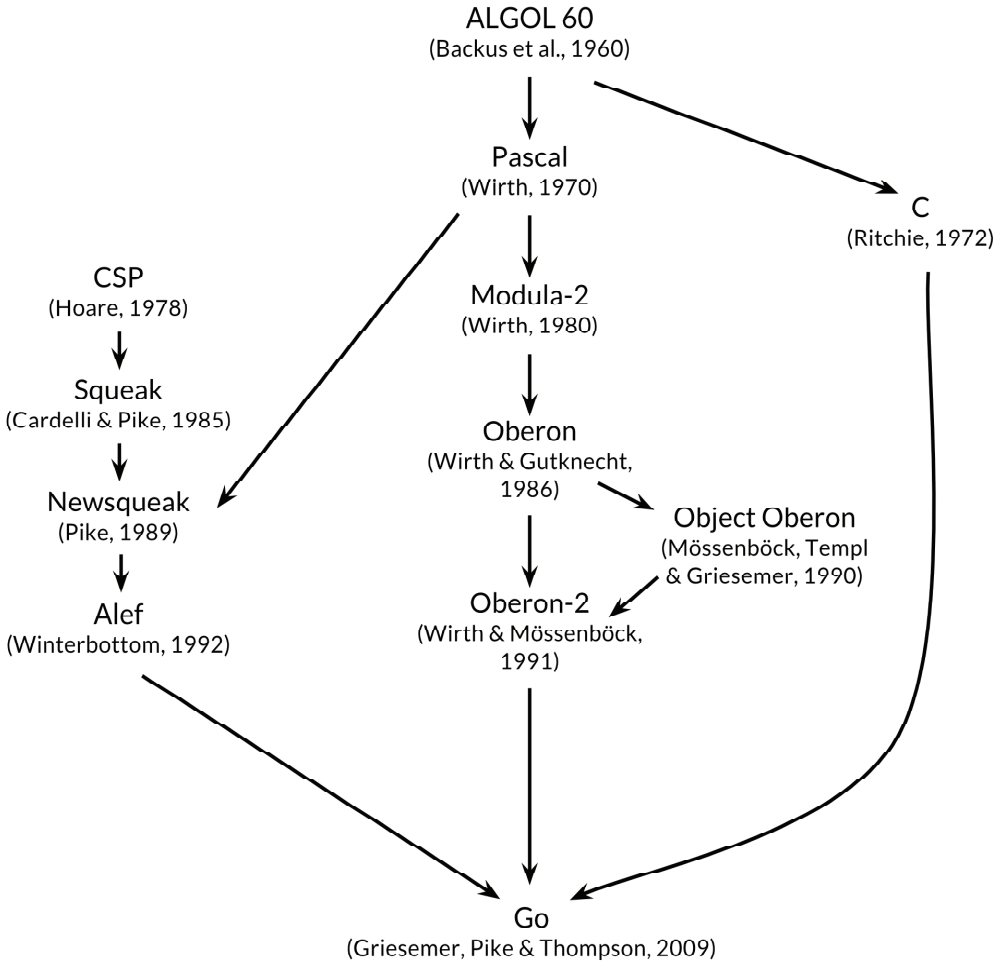


Рисунок 1.1 — Вплив попередніх мов програмування на Go [5]

На дизайн мови програмування Go суттєво вплинули кілька попередніх мов, кожна з яких залишила свій слід у її синтаксисі, концепціях і механізмах,

що продемонстровано на рисунку 1.1. C, безсумнівно, мала найбільший вплив, а саме на такі аспекти:

– синтаксис виразів та операторів керування (умови, цикли тощо);

– базові типи даних;

– модель передачі параметрів за значенням;

– використання вказівників;

– орієнтація на ефективну компіляцію у машинний код;

– взаємодія з низькорівневими абстракціями операційної системи*.

Також мали вплив і мови Ніклауса Вірта, здебільшого саме на модульність і організацію коду:

– Pascal і Modula-2 — надихнули концепцію package;

– Oberon — усунув розмежування між інтерфейсом і реалізацією модуля;

– Oberon-2 — вплинув на синтаксис оголошення пакетів, імпортів і методів.

Унікальний вплив на Go, що не мав великого поширення на інші мови програмування (на противагу попереднім кейсам), мали експериментальні мови Bell Labs на основі CSP, участь в розробці яких приймав безпосередньо Роб Пайк. Вони надихалися ідеями Тоні Хоара (1978), де процеси не мають спільного стану і спілкуються через канали. Нижче наведено їх перелік:

– Squeak — рання реалізація CSP для обробки подій;

– Newsqueak — додав синтаксис, схожий на C, і типізацію як у Pascal, впровадив канали як значення, що можна зберігати і створювати динамічно;

– Alef (Plan 9) — намагався зробити Newsqueak придатним для системного програмування, але відсутність збирача сміття ускладнювала роботу з рівночасністю.

Усе це стало основою для моделі рівночасності Go, що побудована на горутинах і каналах. Більш детальна розповідь про неї знаходиться в підрозділі 1.6.

---

*Саме тому Go іноді називають «C для XXI століття».

Інші менш відомі впливи:

– ідентифікатор `iota` — натхненна мовою APL;

– лексична область видимості та вкладені функції — із Scheme (та багатьох інших мов, що теж це звідти запозичили).

1.1.4 Команда та таймлайн

Роберт Ґрісемер, Роб Пайк і Кен Томпсон почали робити перші начерки цілей нової мови на дошці 21 вересня 2007 року. За кілька днів ці цілі оформилися в конкретний план дій і приблизне уявлення про те, якою має бути мова. Дизайн продовжувався неповний робочий день паралельно з іншою діяльністю. До січня 2008 року Кен приступив до розробки компілятора для випробування ідей; він генерував код на C. Вже до середини року проєкт став повноцінною роботою, і мова була достатньо стабільною для спроби створити повноцінний компілятор. У травні 2008 року Ян Тейлор незалежно почав розробляти фронтенд GCC для Go на основі чернеткової специфікації. Наприкінці 2008 року до проєкту приєднався Расс Кокс, який допоміг вивести мову та бібліотеки з прототипного стану [6].

Перший публічний реліз мови програмування Go відбувся 10 листопада 2009 року, коли Google відкрив вихідний код проєкту як open source [7]. Це дало змогу розробникам по всьому світу ознайомитися з мовою та долучитися до її розвитку. Пізніше, у березні 2012 року, було випущено першу стабільну версію Go 1.0, яка закріпила обіцянку зворотної сумісності та стала основою для подальшого розвитку екосистеми мови [8].

Go дотримується шестимісячного циклу релізів, що забезпечує регулярне впровадження нових функцій та покращень [9].

Одним із ключових оновлень стало впровадження системи модулів у Go 1.11 у серпні 2018 року. Це спростило керування залежностями та дозволило розробникам працювати поза межами GOPATH, що значно покращило організацію проєктів [10].

У березні 2022 року з виходом Go 1.18 було додано підтримку дженериків (узагальнених типів), що було однією з найбільш жаданих функцій з моменту запуску мови. Це дозволило розробникам писати більш гнучкий та повторно використовуваний код без втрати типобезпеки [11].

Останній на момент написання реліз, Go 1.24, вийшов у лютому 2025 року. Він продовжує традицію стабільності та сумісності, водночас впроваджуючи нові можливості та оптимізації [12].

1.2 Синтаксис

Go є мовою загального призначення, що є, як вже було зазначено, компільованою, рівночасною та оснащеною збирачем сміття. Керуючись принципами, що вимагають лаконічності та простоти, автори створили компактний та простий для парсингу C-подібний синтаксис, що полегшує аналіз коду такими автоматичними інструментами, як дебагери, аналізатори залежностей, автоматизовані екстрактори документації, плагіни інтегрованого середовища розробки тощо [6,13]. Класичний приклад продемонстровано нижче (лістинг 1.1).

```go
package main

import "fmt"

func main() {
    fmt.Println("Привіт, Світе!")
}
```

Лістинг 1.1 — Варіант класичної програми "Hello, World!" мовою Go

Не знаючи мови Go, але маючи загальне уявлення про програмування, можна зробити декілька припущень про результат компіляції та виконання програми:

– програма не скомпілюється, натомість компілятор надасть повідомлення про помилку накшталт `Semicolon (';') expected`;

– програма скомпілюється, але в результаті виконання відбудеться виведення в консоль спотвореного тексту через невідповідність кодувань;

– виведення в консоль тексту «Привіт, Світе!».

По-перше, формальний синтаксис мови Go дійсно використовує крапку з комою як символ-термінатор, але в програмному коді більшість з них можна опускати, керуючись двома правилами [13]:

1. Під час розбиття вхідного тексту на токени, крапка з комою автоматично вставляється в потік токенів безпосередньо після останнього токена рядка, якщо цим токеном є:

– ідентифікатор;

– цілочисельний, дійсний, уявний, рунний або рядковий літерал;

– одне з ключових слів break, continue, fallthrough або return;

– один з операторів або розділових символів: ++, –,),], або }.

2. Щоб дозволити запис складних інструкцій в один рядок, крапку з комою можна опустити перед закриваючими дужками) або }.

Через те, що рядок з викликом функції (лістинг 1.1, рядок 6) задовільняє перше правило, крапку з комою там ставити не обов'язково (інші рядки, в свою чергу, також задовільняють ці правила). Більш загально можна зазначити, що крапки з комою в програмному коді на Go наприкінці рядків зазвичай не зустрічаються.

По-друге, файли програмного коду на Go за замовчуванням мають кодування UTF-8*, яке дозволяє зберігати та передавати представлення символів у Unicode більш компактно. За конвенцією, так само кодуються і рядки — вони є незмінною послідовністю байтів, що інтерпретуються як кодові точки Unicode, які в Go також називають «рунами» [5,13]. Виходячи з цього, стає зрозуміло, що завдяки передування конвенції конфігураціям текст відобразиться коректно.

Інші особливості, що варті уваги та продемонстровані в цьому прикладі:

---

*Кен Томпсон та Роб Пайк, що є одними з авторів Go, розробили UTF-8 [14].

1. Кожна програма на Go починається з оголошення пакету; цей приклад не є виключенням (лістинг 1.1, рядок 1). Якщо вказано `main`, це означає, що файл є частиною *головного пакету*, що є обов'язковим для самостійної програми, яка компілюється у виконуваний файл. Саме в пакеті `main` обов'язково повинна міститися функція `main()`, що не приймає аргументів та не повертає значення. Виконання програми починається ініціалізацією пакету `main` та викликом функції `main`; воно завершується з поверненням цього виклику, не чекаючи на завершення інших (не `main`) горутин [13].

2. Go використовує ключове слово `import` для підключення зовнішніх та стандартних бібліотек. У цьому прикладі імпортується стандартний пакет `fmt`, що надає функції для форматованого I/O, зокрема — `Println()`. Можливість зовнішнього доступу до ідентифікатора визначається просто: вона є, якщо його ім'я починається з великої літери [5].

1.2.1 Оголошення

Варто зазначити, що синтакисис оголошення в Go був змінений відносно мови C задля покращення легкості читання коду [15].

1.2.2 Змінні

Оголошення змінної створює одну або кілька змінних, зв'язує відповідні ідентифікатори з ними та призначає кожній тип і початкове значення [13]. Якщо задано список виразів, змінні ініціалізуються цими виразами відповідно до правил операторів присвоєння. В іншому випадку кожна змінна ініціалізується нульовим значенням. Якщо вказано тип, кожна змінна отримає саме цей тип. Інакше змінна набуває типу відповідного значення ініціалізації. Якщо значення — це нетипізована константа, вона спочатку неявно перетворюється на тип за замовчуванням; якщо це нетипізоване булеве значення, воно неявно перетворюється на тип `bool`. Попередньо оголошений ідентифікатор `nil` не можна використовувати для ініціалізації змінної без явного типу.

Оголошена невикористана змінна спричинить помилку компіляції.

У Go можна оголошувати змінні двома способами: за допомогою ключового слова `var` (лістинг 1.2) або скорочено (лістинг 1.3).

```go
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
  i       int
  u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // звернення до map; зацікавлені лише у "found"
```

Лістинг 1.2 — Приклади оголошення змінних за допомогою `var`

```go
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w, _ := os.Pipe()  // os.Pipe() повертає три значення: пару пов'язаних файлів та помилку
_, y, _ := coord(p)   // coord() повертає три значення; зацікавлені лише в координаті y
```

Лістинг 1.3 — Приклади скороченого оголошення змінних

На відміну від звичайного оголошення, скорочене оголошення може перевизначати змінні, якщо вони були оголошені раніше в тому ж блоці (або в списках параметрів, якщо блок є тілом функції) з тим самим типом. Перевизначення можливе лише у скорочених оголошеннях кількох змінних. Перевизначення не створює нову змінну — воно лише присвоює нове значення вже існуючій. Імена змінних, які не є підкресленням (_) і стоять зліва від `:=`, мають бути унікальними. Скорочені оголошення можуть з’являтися лише всередині функцій. У деяких

контекстах, наприклад, в ініціалізаторах для операторів if, for або switch, їх можна використовувати для оголошення локальних тимчасових змінних.

1.2.3 Константи

Константи оголошуються через ключове слово `const` і можуть бути числовими, рядковими або логічними. На відміну від змінних, їх значення не можна змінити після оголошення. Як синтаксичний цукор при оголошенні декількох послідовних констант можна використовувати генератор констант `iota` [13].

1.2.4 Функції

Функції оголошуються за допомогою ключового слова `func` і можуть приймати параметри та повертати значення. Унікальною особливістю Go є можливість повертати кілька значень, що часто використовується для обробки помилок (лістинг 1.4).

```go
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, fmt.Errorf("ділення на нуль")
    }
    return a / b, nil
}
```

Лістинг 1.4 — Приклад оголошення функції

1.2.5 Методи

Методи в Go — це функції, прив'язані до конкретного типу (наприклад, структури). Вони оголошуються з вказівкою приймача (receiver), що дозволяє працювати з даними структури. На лістингу нижче можемо побачити приклад використання методу.

```go
func (p Person) greet() string {
    return fmt.Sprintf("Привіт, я %s!", p.Name)
}
...
greeting := p.greet()
```

Лістинг 1.5 — Приклад оголошення та виклику методу

1.2.6 Умовні оператори

В мові Go існує декілька операторів для створення умов: `if`, `switch` та один з варіантів `for`.

Оператор `if` визначає умовне виконання однієї з двох гілок залежно від значення булевого виразу: якщо вираз оцінюється як `true`, виконується гілка `if`, інакше (якщо присутня) — гілка `else`. Перед умовним виразом може бути просте твердження, яке виконується перед його оцінюванням [13]. Дужки навколо умови не є обов'язковими, а фігурні дужки для тіла умови — обов'язкові.

```go
if x := f(); x > 10 {
    fmt.Println("x більше 10")
} else {
    fmt.Println("x менше або дорівнює 10")
}
```

Лістинг 1.6 — Приклад використання умовного оператора `if`

Оператор `switch` забезпечує множинне розгалуження. Вираз або тип порівнюється з «випадками» (`case`) у середині `switch`, щоб визначити, яку саме гілку виконати [13].

1.2.7 Цикли

У Go є класичний оператор циклу `for`, що має три форми [13]:

1. Проста форма з умовою, є аналогом циклу `while` в інших мовах. Виконує блок коду, доки умова є істинною.

```go
for a < b {
    a *= 2
}
```

Якщо умова відсутня - еквівалентно `true`, що утворює нескінченний цикл (`for {}`).

2. Форма з `for`-clause. Містить три необов'язкові частини, розділені крапкою з комою:

1. Ініціалізація (виконується 1 раз перед початком);
2. Умова (перевіряється перед кожною ітерацією);
3. Пост-умова (виконується після кожної ітерації).

```go
for i := 0; i < 10; i++ {
    f(i)
}
```

З Go 1.22 кожна ітерація має свої окремі змінні (раніше змінні були спільними).

3. Форма з `range`-clause. Використовується для ітерації по масивах, зрізах, рядках (по рунах), мапах, каналах, числових діапазонах (з Go 1.22), функціях-ітераторах (з Go 1.23).

```go
for i, v := range slice {
    // i - індекс, v - значення
}
```

Особливості:

– для рядків ітерація йде по Unicode-символах;

– порядок ітерації мап не гарантований;

– для каналів цикл завершується при закритті каналу;

– числовий діапазон генерує значення від 0 до n-1.

Кожна ітерація створює нові змінні (з Go 1.22). Якщо змінні не оголошені в `range`, вони повинні існувати раніше.

1.2.8 Структури

Структури визначаються через `struct` і дозволяють групувати поля різних типів. Вони використовуються для створення складних типів даних, аналогічно класам в інших мовах, але без спадкування (лістинг 1.7).

```go
type Person struct {
    Name string
    Age  int
}
p := Person{Name: "Alice", Age: 30}
fmt.Println(p.Name) // "Alice"
```

Лістинг 1.7 — Приклад використання структури

1.2.9 Інтерфейси

Інтерфейси визначають набір методів, які мають реалізовувати інші типи. На відміну від багатьох мов, у Go для реалізації інтерфейсу не потрібно явно вказувати це — достатньо, щоб тип мав усі необхідні методи. Інтерфейси визначають поведінку — будь-яка структура з методом `greet()` автоматично реалізує інтерфейс `Greeter` (лістинг 1.8).

```go
type Greeter interface {
    greet() string
}
```

Лістинг 1.8 — Приклад інтерфейсу

1.2.10 Відкладене виконання (`defer`)

Концепт `defer` вперше з'явився саме в мові Go [5] та використовується для відкладеного виконання функції до завершення поточної:

```go
func readFile() {
    file, _ := os.Open("file.txt")
    defer file.Close() // закриє файл після завершення readFile
    // ... читання файлу
}
```

Лістинг 1.9 — Приклад використання `defer`

1.3 Обробка помилок

У Go механізм обробки помилок базується на ідеї представлення помилки як звичайного значення типу `error`. Це дає змогу програмам більш явно і передбачувано реагувати на виникнення несподіваних ситуацій, уникати прихованих виключень і забезпечувати чіткий контроль над логікою виконання. Основні принципи й підходи до обробки помилок у Go можна сформулювати таким чином:

1. Повернення помилки як значення. У Go будь-яка функція або метод, що може завершитися помилкою, зазвичай має останнім повертаним параметром значення типу `error`. В лістингу 1.10 функція `ReadConfig` повертає або заповнену структуру `Config` разом із `nil` (якщо помилки не сталося), або незаповнений `Config` та ненульове значення `error`:

```go
func ReadConfig(path string) (Config, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return Config{}, err
    }
    // ...обробка data...
    return cfg, nil
}
```

Лістинг 1.10 — Приклад повернення помилки

2. Перевірка `err != nil`. Після виклику будь-якого коду, який потенційно може повернути помилку, її необхідно одразу перевірити. Якщо `err` відмінний

від `nil`, логіка програми повинна або перепакувати й повернути цю помилку далі, або обробити її локально й продовжити виконання. Типовий шаблон наведено нижче (див. лістинг 1.11)

```go
result, err := SomeOperation()
if err != nil {
    // обробка помилки (лог, вихід, повернення вище)
    return nil, fmt.Errorf("SomeOperation failed: %w", err)
}
// робота з result
```

Лістинг 1.11 — Типовий шаблон оброблення помилки

3. Створення та форматування помилок. Для створення нового об'єкта помилки використовують функцію `errors.New` або `fmt.Errorf` (див. лістинг 1.12)

```go
import (
    "errors"
    "fmt"
)

func DoSomething(value int) error {
    if value < 0 {
        return errors.New("value must be non-negative")
    }
    if value == 0 {
        return fmt.Errorf("value is zero; expected positive integer")
    }
    // подальша логіка
    return nil
}
```

Лістинг 1.12 — Використання різних способів створення нової помилки

4. Виявлення й порівняння помилок (errors.Is, errors.As). З огляду на те, що помилка — це інтерфейс, часто потрібен механізм для ідентифікації певного типу помилки навіть після її обгортання. Для цього використовують пакунок `errors` із функціями `errors.Is` (повертає `true`, якщо `err` дорівнює `target` або містить

його в ієрархії загортань) та `errors.As` (дає змогу отримати оригінальний об'єкт помилки певного типу, якщо він зустрічається в ланцюгу загортань). На лістингу 1.13 наведено приклад використання функції `errors.Is`.

```go
var ErrNotFound = errors.New("item not found")

func FindItem(id string) (Item, error) {
    // ...
    return Item{}, ErrNotFound
}

func MainFlow() {
    item, err := FindItem("123")
    if err != nil {
        if errors.Is(err, ErrNotFound) {
            // обробка ситуації «елемент не знайдено»
        } else {
            // інші помилки
        }
    }
    // робота з item
}
```

Лістинг 1.13 — Приклад використання `errors.Is`

5. Створення власних типів помилок. Інколи необхідно передати додаткові дані всередині помилки або реалізувати власну логіку. Для цього створюють кастомний тип, який реалізовує інтерфейс `error` (має метод `Error() string`) (приклад наведено на лістингу 1.14).

```go
type ValidationError struct {
    Field   string
    Problem string
}

func (v ValidationError) Error() string {
    return fmt.Sprintf("validation failed: field %q %s", v.Field,
    v.Problem)
}

func Validate(name string) error {
    if name == "" {
        return &ValidationError{Field: "name", Problem: "cannot be
        empty"}
    }
    return nil
}
```

Лістинг 1.14 — Створення власного типу помилок

6. Panic і recover. Механізм `panic/recover` у Go слугує для обробки критичних виняткових ситуацій, які за звичайних умов не слід обробляти через повернення значення `error`. Використовувати `panic` рекомендується тільки в тих випадках, коли подальше виконання програми неможливе або призведе до неконсистентного стану.

```go
func CriticalTask() (err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("CriticalTask panicked: %v", r)
        }
    }()
    // тут може статись panic
    return nil
}
```

Лістинг 1.15 — Типовий шаблон використання в функції

Таким чином, якщо всередині `CriticalTask` виникне `panic`, він буде перехоплений через `recover`, а замість аварійного завершення функція поверне помилку (див. лістинг 1.15).

7. Кращі практики обробки помилок:

– при виявленні помилки краще одразу припинити поточний потік виконання чи повернути її вищій рівень, а не ігнорувати;

– текст помилки має пояснювати, що саме пішло не так, але без надмірної деталізації внутрішньої реалізації;

– не слід багаторазово загортати одну й ту саму помилку без потреби: кожен новий шар має додавати контекст (де і чому сталася помилка);

– не використовувати `panic` для звичайних помилок, тобто перевірочних ситуацій та помилок вводу/виводу завжди слід користуватися поверненням `error`;

– використовувати стандартний пакунок `errors` (Go 1.13+) для упаковки та розпаковки помилок (`errors.Is`, `errors.As`, `errors.Unwrap`);

– написати тести, які перевіряють не лише успішні шляхи виконання, а й сценарії, коли мають виникнути конкретні помилки.

Завдяки таким підходам обробка помилок у Go є явною, передбачуваною та контрольованою. Концепція «помилка як значення» дає можливість гнучко поєднувати різні рівні абстракції, зберігаючи можливість чітко відслідкувати причин виникнення та маршрут поширення помилок у програмі.

1.4 Організація коду

1.4.1 Пакети

Пакети в Go є основним способом організації коду. Кожен файл .go належить до певного пакету, який вказується у першому рядку; наприклад, оголошення пакету *main* відбувається шляхом написання в першому рядку `package main`.

Ключові особливості:

– стандартні пакети (*fmt*, *net/http*, *os* тощо) входять до складу мови;

– імпорт пакетів виконується за допомогою ключового слова `import` (лістинг 1.16);

– експорт ідентифікаторів (доступність з інших пакетів) визначається великою літерою на початку імені;

– ієрархія пакетів відображається у файловій структурі проекту.

```go
import (
    "fmt"        // стандартний пакет
    "math/rand" // пакет з підкаталогу
    "github.com/user/package" // зовнішній пакет
)
```

Лістинг 1.16 — Приклад імпорту різних пакетів

1.4.2 Модулі

Модулі (з Go 1.11+) - це механізм керування залежностями та версіями. Вони визначаються файлом *go.mod*, структура якого наведена на рисунку 1.2. Перша позначка відображає шлях модуля, який є його унікальним ідентифікатором (у поєднанні з номером версії модуля) — зазвичай це розташування репозиторію, з якого інструменти Go можуть завантажити модуль. Шлях модуля стає префіксом імпорту для всіх пакетів, що містяться в модулі [16].

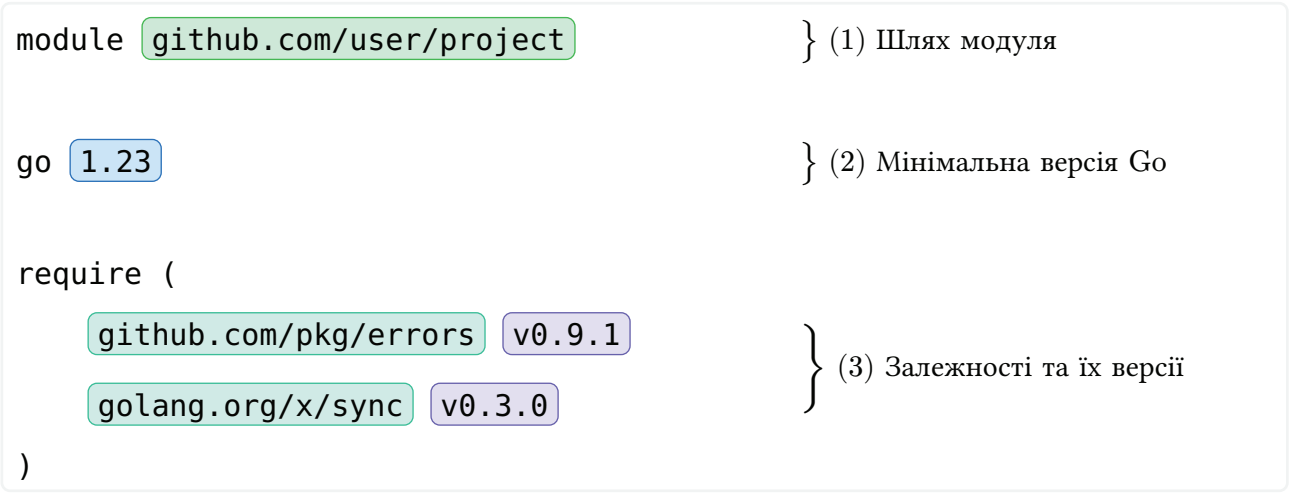


Рисунок 1.2 — Базова структура *go.mod*

Після директиви `go` зазначається версія Go, якою користувалися при написанні цього модуля, яка також є мінімальною версією, що необхідна для його використання (на рисунку 1.2 позначено другим). Директива `require` оголошує мінімально необхідну версію певної залежності модуля. Для кожної вказаної версії модуля команда `go` завантажує відповідний файл `go.mod` і враховує залежності, зазначені в цьому файлі (на рисунку 1.2 позначено третім).

Серед багатого функціоналу команди `go`, про який докладніше розказано нижче, знаходиться також і управління модулями. Так, вона надає можливість ініціалізувати модуль за допомогою `go mod init`, автоматично керувати залежностями за допомогою `go get`, оновлювати залежності за допомогою `go mod tidy` і так далі.

1.5 Інструментарій (команда `go`)

Команда `go` є центральним інструментом для управління початковим кодом на мові програмування Go [17]. Вона забезпечує широкий спектр функцій, включаючи компіляцію, тестування, управління залежностями та модулями, форматування коду та багато іншого. Цей інструмент є невід'ємною частиною екосистеми Go і сприяє ефективній розробці програмного забезпечення.

Команда go підтримує низку підкоманд, кожна з яких виконує специфічну задачу:

– `go build` відповідає за компіляцію вказаних пакетів та їхніх залежностей без автоматичного встановлення результату. При запуску в кореневій теці модуля або в межах пакета вона збирає всі необхідні файли, перевіряє їх на синтаксичні та типові помилки, після чого створює виконуваний файл (або об'єктні файли у разі пакетів-бібліотек). Якщо запустити `go build` у корені модуля з пакетом `main`, утворюється виконуваний файл із назвою каталогу (або іменем, зазначеним через прапорець `-o`). У разі помилок під час компіляції команда виводить детальні повідомлення, що дозволяють швидко знайти та виправити некоректні ділянки коду.

– `go run` призначена для швидкого запуску Go-програм без необхідності явної генерації проміжного виконуваного файлу в робочому каталозі. При використанні `go run` вказується один чи кілька Go-файлів (наприклад, `go run main.go`), після чого інструментарій автоматично компілює їх у тимчасовий виконуваний файл у кеші, одразу ж запускає його й видаляє після завершення виконання. Це зручно під час розробки або тестування невеликих утиліт, оскільки дозволяє одразу перевірити поведінку програми без створення додаткових файлів у проєкті.

– `go test` відповідає за запуск модульних тестів, описаних у файлах із суфіксом `_test.go`. Вона знаходить у пакеті усі функції, які відповідають шаблону `TestXxx(t testing.T)`, автоматично виконує їх у відокремленому середовищі й повідомляє про успішність або невдачу кожного тесту. Також `go test` підтримує додаткові режими, зокрема бенчмаркинг (функції `BenchmarkXxx`) та приклади (функції `ExampleXxx`). За потреби можна передавати різні прапорці для детальнішого виведення (наприклад, `-v` для виведення всіх повідомлень) або для обмеження тестів по назві та інших параметрах.

– `go fmt` автоматично форматує вихідні файли Go відповідно до офіційного стандарту стилю, визначеного в інструменті `gofmt`. При запуску в каталозі з вихідним кодом вона перебирає всі `.go` файли і змінює відступи, вирівнювання елементів, розташування дужок і пробілів так, щоб код відповідав єдиному кано-

ну оформлення. Це гарантує узгодженість стилю в межах команди розробників і полегшує читання та підтримку коду. Виконуючи `go fmt ./...`, можна відформатувати всі файли модуля або робочої теки рекурсивно.

– `go get` використовується для завантаження та встановлення зовнішніх пакетів і їхніх залежностей у середовище розробника. Вона додає потрібні модулі до файлу `go.mod` (якщо він існує), завантажує відповідні версії пакетів і зберігає їх у кеш (`GOPATH/module-cache`). Крім того, `go get` може оновлювати вже існуючі залежності до вказаної версії (наприклад, `go get example.com/pkg@v1.2.3`), автоматично переглядаючи сумісність та фіксуючи зміни у `go.mod` та `go.sum`. Сучасна рекомендація — використовувати `go get` переважно для управління залежностями всередині модуля, а не для встановлення глобальних утиліт.

– `go mod` відповідає за керування модулями та їхніми залежностями. Вона об'єднує низку підкоманд:

  – `go mod init` створює файл `go.mod` і ініціалізує новий модуль у поточному каталозі.

  – `go mod tidy` видаляє невживані залежності та додає відсутні, необхідні для успішної компіляції.

  – `go mod vendor` копіює всі залежності в директорію `vendor` для створення відокремленого каталогу з локальними бібліотеками.

  – `go mod verify` перевіряє цілісність завантажених модулів за контрольними сумами з файлу `go.sum`.

Ці інструменти забезпечують точний контроль версій, синхронізацію залежностей і відтворюваність збірок у Go-проєктах.

– `go install` призначена для компіляції пакета (або пакета `main`) та встановлення отриманого виконуваного файлу в директорію, визначену змінною середовища `GOBIN` (або за замовчуванням `$GOPATH/bin`). При виклику `go install ./...` відбувається збірка всіх пакетів у поточному модулі та встановлення результатів. Якщо вказувати версію під час встановлення (наприклад, `go install example.com/cmd@latest`), інструмент завантажить вихідні коди з відда-

леного репозиторію, скомпілює і встановить виконуваний файл безпосередньо у GOBIN.

– go list надає інформацію про пакети та модулі в структурованому форматі. За замовчуванням вона виводить список імпортних шляхів знайдених пакетів, але може також згенерувати JSON із докладними метаданими (через прапорець -json), такими як версія модуля, залежності, файли сорців та інші атрибути. Це корисно для автоматизації, інтеграції з CI/CD або створення власних інструментів аналізу коду.

– go doc відображає документацію для пакета або окремого символу (функції, типу, змінної) безпосередньо в терміналі. Якщо вказати лише ім'я пакета (go doc fmt), відобразяться заголовкові коментарі з файлу doc.go, а якщо додати символ (go doc fmt.Printf), буде виведено опис саме цієї функції. Це полегшує швидкий перегляд документації без необхідності відкривати веб-браузер або виходити з командного рядка.

– go clean використовується для видалення артефактів компіляції, об'єктних файлів і кешованих даних. Виконуючи go clean у межах пакета, інструмент видалить згенеровані файли (наприклад, .o, .a, .exe) та тимчасові каталоги. Додаткові прапорці (-cache, -testcache, -modcache) дозволяють очищати відповідні кеші інструментів (build cache, test cache, module cache). Це допомагає позбутися застарілих даних і гарантувати, що наступна збірка відбудеться «з нуля».

– go env виводить значення змінних середовища, пов'язаних із середовищем Go, таких як GOPATH, GOROOT, GOOS, GOARCH та інші. З її допомогою можна перевірити, як налаштовано робоче середовище, і визначити шляхи до каталогів кешу, модулів, пакетів та інструментів. Вивід може бути використаний у скриптах автоматизації або для діагностики проблем із конфігурацією.

– go vet здійснює статичний аналіз коду для виявлення потенційних помилок, невідповідностей або поганих практик, які компілятор пропустить. Серед перевірок — невикористані змінні, неправильне форматування рядків при

виклику `Printf`, невідповідність специфікаторів форматування та типів тощо. У результаті виконання `go vet` видає попередження, що допомагають виправити помилки до етапу виконання програм.

– `go generate` призначена для автоматичної генерації допоміжного коду на основі вбудованих директив у вихідних файлах Go. Розробник додає в коментарі до файлу рядок виду `//go:generate <команда>`, наприклад, виклик `stringer` або власного генератора. При виконанні `go generate ./...` інструмент проходить усі файли, знаходить такі директиви й виконує відповідні зовнішні утиліти. Це полегшує створення кодогенерованих елементів (наприклад, реалізацій інтерфейсів, енумів, схем API) без ручного втручання.

– `go version` виводить поточну версію інстальованого компілятора Go, а також інформацію про операційну систему та архітектуру, на яких він запущений. Наприклад, виконання видасть рядок `go version go1.20.4 linux/amd64`. Це корисно для перевірки, чи відповідає середовище вимогам проєкту, і для діагностики ситуацій, коли код може залежати від конкретної версії Go.

– `go work` створює та керує робочими просторами, які дозволяють об'єднувати кілька модулів у єдине середовище розробки. Файл `go.work` містить список локальних модулів, щоб інструмент `go` використовував їх замість публічних версій з проксі-серверів. Це зручно при розробці великих додатків, які складаються з кількох окремих модулів, що розвиваються одночасно. За допомогою `go work init`, `go work use` та інших підкоманд можна додавати чи видаляти модулі з робочого простору та оновлювати конфігурацію залежностей.

1.6 Модель рівночасності

У мові Go концепція рівночасності базується на передачі даних через комунікацію, а не на доступі до спільної пам'яті. Вона реалізує рівночасність через легковагові *горутини* (goroutines) та *канали* (channels), засновані на принципах моделі CSP. Замість того, щоб захищати спільні змінні м'ютексами та іншими примітивами синхронізації, Go пропонує передавати значення між горутинами за

допомогою каналів, гарантуючи, що в будь-який момент часу до даних має доступ лише одна горутина. Такий підхід унеможливлює виникнення *гонок даних* (data races) та спрощує мислення про паралельні обчислення: комунікація сама по собі виступає механізмом синхронізації [18].

В більшості популярних мов програмування рівночасність та паралелізм досягаються за допомогою потоків ОС та спільної пам'яті (shared memory). Так це відбувається, наприклад, у Java та C++. Проблема в тому, що потоки ОС є доволі важкими (1–2 МБ на потік) і вимагають ручного керування синхронізацією, а використання спільної пам'яті часто призводить до складних у виявленні race conditions.

Інші популярні мови мають свої особливості реалізації рівночасності та паралелізму. Наприклад, Python має обмежену рівночасність через GIL (Global Interpreter Lock), який блокує виконання потоків для CPU-зв'язаних задач. Навіть з модулем `asyncio`, Python ефективний лише для I/O-операцій, тоді як Go забезпечує справжній паралелізм на багатоядерних системах через `GOMAXPROCS`. Rust використовує модель на основі `async/await` для асинхронного коду, що потребує явного використання бібліотек (наприклад, `tokio`). Хоча Rust ефективний для системного програмування, його підхід менш інтуїтивний для простих конкурентних задач порівняно з горутинами Go.

Заслуговує уваги модель, що наявна в Erlang - доволі старій функціональній мові програмування, популярній у сфері телекомунікацій. Erlang, як і Go, використовує легковагові *процеси* (подібні до горутин, навіть легші), що існують всередині віртуальної машини Erlang, і *обмін повідомленнями* (message passing). Однак Erlang не має каналів — кожен процес має власну «поштову скриньку», а обмін даними відбувається через асинхронні повідомлення. Перевага Go — у простоті синтаксису для CSP-моделі, тоді як Erlang вимагає більш функціонального підходу.

Горутина — це легковаговий потік виконання, який запускається ключовим словом `go` перед викликом функції або методу. Стек горутини починається з

невеликого розміру й автоматично збільшується в міру необхідності, що робить створення та контекстну зміну горутин вельми економними з погляду пам'яті. У разі блокування однієї горутини (наприклад, при очікуванні I/O) інші продовжують виконуватись, оскільки рантайм Go розподіляє горутини по потоках ОС.

Канали у Go є засобом передачі значень між горутинами й водночас гарантують синхронізацію. Небуферизований канал блокує відправника до отримувача та навпаки, що дозволяє природним чином реалізувати очікування завершення операції. Буферизований канал з фіксованою місткістю дає змогу відправнику продовжити роботу до заповнення буфера, після чого він чекає, поки споживач звільнить місце. Оператор `select` розширює можливості каналів, дозволяючи одній горутині одночасно очікувати на кілька каналів і обирати ту гілку обробки, подія в якій відбулася першою.

Патерн семафора реалізується через канал з обмеженою пропускною здатністю: перед обробкою запиту горутина надсилає умовний "токен" у канал, а після завершення — забирає його назад, таким чином обмежуючи одночасну кількість активних обробників. Альтернативний і більш ресурсозбережний підхід полягає у створенні фіксованої кількості обробних горутин, які нескінченно приймають завдання з одного спільного каналу, виконують їх і лише потім повертаються до очікування наступного завдання.

Ідеї рівночасності у Go також застосовуються до паралелізації обчислень на багатоядерних системах. Якщо завдання можна розбити на незалежні підзадачі, їх можна виконувати одночасно в різних горутинах із відправленням сигналів про завершення за допомогою каналу. Після запуску всіх підзадач основна горутина вивільняє канал, очікуючи рівно стільки сигналів, скільки підзадач було створено. Значення кількості ядер часто отримують через `runtime.GOMAXPROCS`, що дозволяє автоматично адаптувати масштаб паралелізації до ресурсів машини.

Завдяки вбудованим у мову механізмам горутин і каналів підхід Go до рівночасності є одночасно потужним і простим у використанні. Він дозволяє створювати масштабовані та безпечні багатопотокові програми без необхідності

оперувати низькорівневими примітивами синхронізації, зводячи весь контроль за спільним доступом до виразної моделі передачі повідомлень.

Висновки за розділом 1

В першому розділі кваліфікаційної роботи було проаналізовано особливості мови програмування Go в розрізі безпечної розробки вебдодатків. Можна зазначити, що Go з'явилась як відповідь на поєднання апаратного прогресу (багатоядерність, мережевість) та інженерних викликів (масштаб кодових баз, швидкість розробки), коли існуючі мови не могли забезпечити одночасно простоту, ефективність і безпеку. Цей набір передумов призвів до того, що у вересні 2007 року Роберт Ґрісемер, Роб Пайк і Кен Томпсон почали працювати над концепціями, що згодом оформилися в мову Go, офіційно опубліковану 10 листопада 2009 року.

Синтаксис Go є мінімалістичним, але потужним, з акцентом на читабельність та однозначність. Відсутність зайвих конструкцій (наприклад, класів або спадкування) компенсується простими механізмами, такими як структури, інтерфейси та функції. Окремої уваги заслуговує концепт «помилки як значення» та відсутність виключень (exception) (натомість — механізм `panic/recover`), що вирізняє цю мову з-поміж аналогічних.

Go стала однією з перших «нових» мов, яка поєднала у собі швидкість компіляції, зручність динамічних мов та вбудовані механізми рівночасності й управління пам'яттю, а також уніфікований та потужний інструментарій, що робить мову привабливим інструментом для розробки ефективних і безпечних вебдодатків.

Розділ 2. ДОСЛІДЖЕННЯ ВРАЗЛИВОСТЕЙ ВЕБДОДАТКІВ

Вебдодаток (або вебзастосунок) — розподілений додаток, в якому клієнтом виступає браузер, а сервером — вебсервер. Браузер може бути реалізацією так званих тонких клієнтів — логіка додатку зосереджується на сервері, а функція браузера полягає переважно у зображенні інформації, завантаженої мережею з сервера, і передачі назад даних користувача. Однією з переваг такого підходу є той факт, що клієнти не залежать від конкретної операційної системи користувача, тому вебдодатки є міжплатформовими сервісами. Унаслідок цієї універсальності й відносної простоти розробки вебдодатки стали широко популярними в кінці 1990-х — початку 2000-х років [19]. В цьому розділі досліджено типові загрози безпеці вебдодатків згідно з OWASP Top 10 та у відповідності до них проаналізовано рекомендації OWASP Go Secure Coding Practices.

2.1 Аналіз OWASP Top 10

OWASP Top 10 — це авторитетний та широко визнаний перелік десяти найкритичніших ризиків безпеки вебдодатків, який публікується міжнародною некомерційною організацією OWASP. Цей документ слугує стандартом підвищення обізнаності для розробників і фахівців з безпеки, допомагаючи їм ідентифікувати та усувати найпоширеніші вразливості у вебдодатках.

Вперше опублікований у 2003 році, OWASP Top 10 регулярно оновлюється на основі аналізу даних про вразливості, зібраних з усього світу. Останнє оновлення на момент написання роботи відбулося у 2021 році — нажаль, заплановане оновлення в 2025 році ще не було опубліковано. Цей перелік не лише відображає найпоширеніші вразливості, але й надає рекомендації щодо їх запобігання, сприяючи впровадженню безпечних практик розробки програмного забезпечення [20].

Кожна категорія в OWASP Top 10 описує конкретний тип вразливості, її потенційні наслідки та методи захисту (див. табл. 2.1). Цей перелік використовується як основа для навчання, аудиту безпеки та розробки політик безпеки в

організаціях. Багато стандартів та нормативних актів, таких як MITRE, PCI DSS (Payment Card Industry Data Security Standard), DISA (Defense Information Systems Agency), FTC (Federal Trade Commission) посилаються на OWASP Top 10 як на еталон у сфері безпеки вебдодатків [21].

Таблиця 2.1

Короткий опис кожної категорії OWASP Top 10 (2021)

OWASP Top 10 (2021)	**Короткий опис**
A01:2021 – Broken Access Control	Недостатній контроль доступу
A02:2021 – Cryptographic Failures	Помилки в реалізації криптографії
A03:2021 – Injection	Ін'єкції, зокрема SQL, NoSQL, OS, LDAP
A04:2021 – Insecure Design	Небезпечна архітектура та дизайн
A05:2021 – Security Misconfiguration	Помилки конфігурації
A06:2021 – Vulnerable and Outdated Components	Використання вразливих або застарілих компонентів
A07:2021 – Identification and Authentication Failures	Збої автентифікації та ідентифікації
A08:2021 – Software and Data Integrity Failures	Порушення цілісності ПЗ та даних
A09:2021 – Security Logging and Monitoring Failures	Проблеми з журналюванням та моніторингом
A10:2021 – Server-Side Request Forgery (SSRF)	Підробка серверних запитів

OWASP Top 10 залишається ключовим ресурсом для розробників, тестувальників та фахівців з безпеки, забезпечуючи актуальну інформацію про найнебезпечніші вразливості та способи їх уникнення. Використання цього переліку

сприяє підвищенню загального рівня безпеки вебдодатків та захисту користувацьких даних.

2.2 Аналіз OWASP Go Secure Coding Practices Guide

OWASP Go Secure Coding Practices Guide [22] — це спеціалізований посібник, створений для розробників, які використовують мову програмування Go у веброзробці. Його основна мета — допомогти уникнути поширених помилок безпеки, надаючи практичні рекомендації та приклади коду, що базуються на принципах безпечного програмування.

Цей посібник був спочатку розроблений командою Checkmarx Security Research Team і пізніше переданий до OWASP Foundation. Він адаптує загальні принципи з OWASP Secure Coding Practices Quick Reference Guide до специфіки мови Go, забезпечуючи розробників конкретними прикладами та рекомендаціями, як уникнути вразливостей у своїх додатках. Нижче відображені основні відомості щодо тем, які відповідають окремим розділам посібника.

2.2.1 Валідація вхідних даних

Невірна або відсутня валідація вхідних даних може призвести до різноманітних атак, таких як SQL-ін'єкції, XSS та інші. Усі вхідні дані повинні бути перевірені на відповідність очікуваному формату, типу та діапазону значень.

Усі вхідні дані слід розглядати як потенційно небезпечні й обов'язково перевіряти на довіреній стороні — на сервері. Перший етап валідації полягає в застосуванні whitelisting (дозволених символів або форматів) та boundary-checking (перевірці довжини рядків чи діапазонів числових значень). Наприклад, користувацький ввід можна обрізати та привести до потрібного регістру перед подальшою обробкою (див. лістинг 2.1).

```go
trimmed := strings.TrimSpace(input) // видалити зайві пробіли
lower   := strings.ToLower(trimmed) // перевести до нижнього регістру
```

Лістинг 2.1 — Підготовка користувацького вводу до обробки

Додатково рекомендується перевіряти формат числових значень (див. лістинг 2.2) та складніші шаблони за допомогою регулярних виразів (див. лістинг 2.3).

```go
n, err := strconv.ParseInt(s, 10, 64)
if err != nil {
    return fmt.Errorf("невірний формат числа: %v", err)
}
```

Лістинг 2.2 — Перетворення рядка в ціле число

```go
re := regexp.MustCompile(`^[a-zA-Z0-9]+$`)
if !re.MatchString(lower) {
    return errors.New("дозволені лише латинські літери та цифри")
}
```

Лістинг 2.3 — Перевірка за регулярним виразом (тільки латинські літери та цифри)

Особливу увагу слід звернути на коректність UTF-8: неприпустимі “overlong” чи розширені послідовності можуть призводити до вразливостей (перевірка наведена у лістингу 2.4).

```go
if !utf8.ValidString(s) { // використовується частина
стандартної бібліотеки "unicode/utf8"
    return errors.New("невірна UTF-8 послідовність")
}
```

Лістинг 2.4 — Перевірка коректності кодування рядка в UTF-8

Після успішної валідації варто виконати post-validation actions: відкинути невідповідні дані з повідомленням про помилку або, за потреби, автоматично скоригувати ввід (наприклад, обрізати зайві символи). Водночас важливо зберігати журнали подій без витоку чутливої інформації і, якщо дані надходять із зовнішніх джерел, додавати перевірки цілісності (hash totals, referential integrity тощо). Для комплексних сценаріїв валідації можна звернутися до сторонніх бібліотек, таких

як `github.com/go-playground/validator`, яка дозволяє задавати правила валідації через теги структур [22].

2.2.2 Кодування вихідних даних

Усі вихідні дані слід коректно кодувати відповідно до контексту, у який вони вставляються, аби уникнути ін'єкцій та XSS-атак. Для HTML-контексту достатньо застосувати функції екранування з пакета `html` (див. лістинг 2.5) або – ще краще – використати шаблонізатор, який робить це автоматично (див. лістинг 2.6).

```go
escaped := html.EscapeString(userInput)  // перетворить < на &lt;, > на &gt; тощо
```

Лістинг 2.5 — Функція екранування

```go
tmpl := template.Must(template.New("T").Parse(`{{define "T"}}<div>{{.}}</div>{{end}}`))
tmpl.ExecuteTemplate(w, "T", userInput)
```

Лістинг 2.6 — Шаблонізатор

Якщо потрібно вставити дані в JavaScript-контекст (наприклад, у скрипт всередині сторінки), слід використати `template.JSEscaper` або JSON-серіалізацію, щоб уникнути розриву рядка чи виконання довільного коду (див. лістинг 2.7).

```go
data, _ := json.Marshal(userData)
tmpl := template.Must(template.New("JS").Parse(`<script>var data = {{.}};</script>`))
tmpl.ExecuteTemplate(w, "JS", template.JS(data))
```

Лістинг 2.7 — Маршалінг JSON для шаблону JavaScript

Для URL-контексту (збирання параметрів запиту) слід використовувати `url.QueryEscape`, а для CSS – `template.CSS`. Кожен з цих енкодерів гарантує, що спеціальні символи не змінять синтаксис об'єкта чи впровадять небезпечний код.

Ще одна популярна вразливість — SQL-ін'єкція — існує здебільшого завдяки поганій практиці конкатенації рядків при створенні запиту. Це вирішується за допомогою заповнювачів (placeholder) в заготовлених запитах (prepared statements) (лістинг 2.8).

```go
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = " + customerId
row, _ := db.QueryContext(ctx, query)
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = ?"
stmt, _ := db.QueryContext(ctx, query, customerId)
```

Лістинг 2.8 — Запобігання SQL-ін'єкціям

2.2.3 Автентифікація та управління паролями

Усі механізми автентифікації мають виконуватися на довіреній стороні (сервері), а не у клієнтському коді чи в самих ресурсах, що потребують захисту. Рекомендується використовувати перевірені та підтримувані фреймворки або централізовані сервіси автентифікації, перевіривши їхній вихідний код на відповідність безпечним практикам. Облікові дані для доступу до зовнішніх систем також слід шифрувати та зберігати у захищених конфігураційних файлах на сервері, а не в коді програми.

Зберігання паролів у відкритому вигляді чи за допомогою застарілих або власних хеш-функції суворо заборонено. Одним із найпростіших і найбільш надійних рішень є пакет `bcrypt` із `golang.org/x/crypto`. Виходячи з цього, в БД поітрібно зберігати не сам пароль, а його хеш. Приклад генерації наведено в лістингу 2.9.

```go
password := []byte("UserSuppliedPassword")
hash, err := bcrypt.GenerateFromPassword(password, bcrypt.DefaultCost)
```

Лістинг 2.9 — Генерація хешу з витратами за замовчуванням

Перевірка введеного пароля відбувається проти збереженого хешу (для лістингу 2.9 це буде виглядати як `bcrypt.CompareHashAndPassword(hashFromDB, password)`).

Щоб унеможливити використання надто простих або повторно вжитих паролів, слід встановити вимоги до складності (літери, цифри, спецсимволи) та мінімальної довжини (не менше 8–16 символів), що можна робити за допомогою регулярних виразів. Критичні системи можуть диктувати більш жорсткі політики та регулярну зміну паролів із забороною повторного вживання. Механізм відновлення пароля повинен:

– Генерувати тимчасовий посилання чи пароль із коротким часом життя,

– Відправляти його лише на попередньо верифіковану адресу електронної пошти,

– Забезпечувати обов'язкову зміну тимчасового пароля при наступному вході.

Облікові дані для входу слід надсилати тільки через захищені HTTPS-з'єднання і тільки методом POST, щоби уникнути логування в URI та історії браузера. У HTML-формі (лістинг 2.10) поле `<input type="password" autocomplete="off">` перекриває автозаповнення, а єдине повідомлення про невдалу автентифікацію має бути загальним: «Невірне ім'я користувача та/або пароль», без уточнення, що саме не співпало.

```html
<form method="post" action="https://example.com/signin"
autocomplete="off">
  <label>Користувач <input type="text" name="username"></label>
  <label>Пароль <input type="password" name="password"></label>
  <input type="submit" value="Увійти">
</form>
```

Лістинг 2.10 — HTML-форма для провадження паролю

2.2.4 Управління сесіями

Управління сесіями вимагає надійної генерації й зберігання ідентифікаторів сесії на сервері та захищеної передачі їх клієнту за допомогою HTTP-cookie.

Перш за все, кожна сесія повинна мати унікальний, криптографічно стійкий ідентифікатор. Для цього можна використати `crypto/rand` (див. лістинг 2.11).

```go
// Генеруємо 32-байтовий токен сесії
b := make([]byte, 32)
if _, err := rand.Read(b); err != nil {
    return "", err
}
sessionID := hex.EncodeToString(b)  // 64-символьний hex-рядок
```

Лістинг 2.11 — Генерація токену сесії за допомогою `crypto/rand`

Потім цей ідентифікатор передається клієнту в HTTP-cookie з суворими прапорами безпеки (див. лістинг 2.12).

```go
cookie := &http.Cookie{
  Name:     "session_id",
  Value:    sessionID,
  Path:     "/",
  Domain:   "example.com",
  Expires:  time.Now().Add(24 * time.Hour),
  HttpOnly: true,              // забороняє доступ з JavaScript
  Secure:   true,              // передається лише по HTTPS
  SameSite: http.SameSiteLaxMode, // захищає від CSRF
}
http.SetCookie(w, cookie)
```

Лістинг 2.12 — Формування cookie

При завершенні сесії (logout) слід не лише видалити cookie на клієнті, а й знищити запис у сховищі сесій: Таким чином забезпечується те, що сесії генеруються передбачувано, передаються тільки через захищене з'єднання й знищуються одразу після виходу користувача (див. лістинг 2.13).

```go
// На сервері відбувається видалення sessionID з Redis чи БД
store.Delete(sessionID)

// На клієнті cookie інвалідується з минулим Expires або MaxAge=-1
expired := &http.Cookie{
  Name:     "session_id",
  Value:    "",
  Path:     "/",
  Expires:  time.Unix(0, 0),
  MaxAge:   -1,
  HttpOnly: true,
  Secure:   true,
}
http.SetCookie(w, expired)
```

Лістинг 2.13 — Видалення та інвалідація cookie

2.2.5 Контроль доступу

Усі перевірки прав доступу мають виконуватись на довіреній стороні – на сервері – і базуватись на принципі *відмови за замовчуванням*: якщо не вдалось однозначно підтвердити дозвіл, у доступі слід відмовляти. Для цього корисно використовувати централізовані ППЗ або функції-обгортки (wrappers), які перед виконанням операції перевіряють, чи має поточний користувач необхідні ролі чи права. Неправильна реалізація контролю доступу може дозволити користувачам отримувати доступ до ресурсів, на які вони не мають прав - необхідно забезпечити, щоб всі запити перевірялися на наявність відповідних прав доступу.

2.2.6 Криптографічні практики

У криптографії чітко розрізняють хешування і шифрування. Функція хешування — одностороння функція, що приймає дані та повертає вихідний рядок фіксованої довжини. Надійні та перевірені алгоритми (BLAKE2, SHA-256) мінімізують ймовірність колізій і не дозволяють відновити початкові дані з хеша.

Для захисту паролів слід застосовувати *сіль* і повільні односторонні функції з достатньою ітеративністю.

При цьому кожен запис у БД має містити як сіль, так і результат хешування. Ще безпечніше покладатися на перевірені адаптивні алгоритми, як-от bcrypt, Argon2 або scrypt, доступні через `golang.org/x/crypto`. Для безпечної генерації випадкових чисел, ідентифікаторів сесій або солі не слід використовувати `math/rand`, оскільки він детермінований і легко прогнозується. Замість цього застосовується криптографічно безпечне хешування з `crypto/rand`. Приклад безпечної генерації та збереження перевірочних сум наведено в розділі 3.

Коли потрібно зберегти чи передати чутливі дані так, щоб їх можна було відновити, використовують шифрування. На лістингу 2.14 наведено приклад застосування симетричного шифру із автентифікацією.

Go

```
import (
  "crypto/aes"
  "crypto/cipher"
  "crypto/rand"
  "io"
)

func encrypt(plaintext, key []byte) ([]byte, error) {
  block, _ := aes.NewCipher(key)
  aead, _ := cipher.NewGCM(block)
  nonce := make([]byte, aead.NonceSize())
  io.ReadFull(rand.Reader, nonce)
  return aead.Seal(nonce, nonce, plaintext, nil), nil
}

func decrypt(data, key []byte) ([]byte, error) {
  block, _ := aes.NewCipher(key)
  aead, _ := cipher.NewGCM(block)
  n := aead.NonceSize()
  return aead.Open(nil, data[:n], data[n:], nil)
}
```

Лістинг 2.14 — Спосіб застосування AES-GCM

2.2.7 Обробка помилок та логування

Обробка помилок в Go в цьому посібнику розглянута дещо поверхнево, і детальний розгляд в пункті 1.3 здебільшого повторює та доповнює цей матеріал.

Натомість, про логування сказано, що воно завжди має здійснюватися на рівні самого застосунку й не повинно залежати від конфігурації сервера. Усі записи логу повинні виконуватися головною рутинною (master routine) на довіреній системі, а розробники мають переконатися, що в логах не містяться конфіденційні дані (наприклад, паролі, інформація про сесії, деталі системи тощо), а також немає ніякої відлагоджувальної інформації чи стек-трейсів. Крім того, логування повинно охоплювати як успішні, так і неуспішні події безпеки, із наголосом на важливих даних про події.

До найважливіших даних про події найчастіше належать:

– Збої валідації введених даних.
– Спроби автентифікації, особливо невдалі.
– Помилки контролю доступу.
– Ознаки спроб втручання, включаючи несподівані зміни стану даних.
– Спроби під’єднання з недійсними або простроченими токенами сесії.
– Виняткові ситуації в системі.
– Адміністративні функції, зокрема зміни налаштувань безпеки.
– Невдалі TLS-з’єднання з бекендом та збої криптографічних модулів.

Неналежна обробка помилок може розкривати внутрішню інформацію про систему, що допомагає атакуючим, тож необхідно уникати виведення детальних повідомлень про помилки користувачам.

2.2.8 Захист даних

У посібнику підкреслюється, що всі компоненти системи повинні працювати за принципом найменших привілеїв, надаючи користувачам і сервісам лише мінімально необхідні права, а тимчасові та кешовані файли з чутливою інформа-

цією негайно видаляти або пересувати в захищене сховище. Не менш важливо ретельно видаляти з коду будь-які приховані ключі або коментарі з API-токенами, а також уникати передачі паролів і токенів через URL-рядки, щоб вони не залишалися в історії браузера чи логах сервера. Вся передача чутливих даних повинна відбуватися виключно через HTTPS з використанням методу POST, а дані «у спокої» — зберігатися у зашифрованому вигляді за допомогою автентифікованих алгоритмів, наприклад NaCl Secretbox або AES-GCM. Крім того, для виявлення будь-яких спроб модифікації логів або важливих файлів слід регулярно перевіряти контрольні суми та хеші сховищ.

2.2.9 Безпека комунікацій

У посібнику наголошується, що всі канали зв'язку — клієнт-сервер, сервер-база даних та внутрішня комунікація між сервісами — мають бути захищені шифруванням для забезпечення конфіденційності, автентичності та цілісності даних і запобігання MITM*-атакам. Крім того, усі HTTP-запити мають містити чіткий `Content-Type` з `charset=utf-8`, а з заголовків слід видаляти будь-яку чутливу інформацію при зверненні до зовнішніх ресурсів, щоб запобігти її витоку.

2.2.10 Системна конфігурація

Неправильна або надлишкова конфігурація компонентів відкриває численні вектори атак. Слід видаляти або відключати невикористовувані служби та функціонал, налаштовувати тільки мінімально необхідні опції в ОС та веб-сервері, а також ізолювати середовища розробки від продакшн-мережі та застосовувати управління змінами для коду та налаштувань.

2.2.11 Безпека БД

Запобігти несанкціонованому доступу допомагають: безпечна інсталяція СУБД із відключенням дефолтних акаунтів і сервісів, встановлення складних

*Man-in-the-middle.

паролів для адміністративних облікових записів і видалення тестових баз та невикористовуваних процедур; обмеження прав користувачів до мінімально необхідних (принцип least privilege); а також параметризовані запити та підготовлені вирази замість конкатенації рядків.

2.2.12 Управління файлами

Щоби уникнути небезпечного завантаження чи доступу до сторонніх файлів, слід дозволяти завантаження тільки автентифікованим користувачам, перевіряти MIME*-тип (whitelisting) та розміщувати завантажені дані поза кореневою директорією веб-додатка в каталозі без прав на виконання. Додатково варто сканувати файли на віруси й обмежувати динамічні редиректи лише перевіреними відносними шляхами.

2.2.13 Управління пам'яттю

Хоч Go і автоматично керує пам'яттю, необхідно виконувати власні перевірки меж буферів (щоб уникнути panic) та завжди закривати ресурси (файлові дескриптори, БД тощо) за допомогою `defer`, а не покладатися лише на збирач сміття. Це запобігає витокам ресурсів і непередбачуваним помилкам під час високого навантаження.

2.2.14 Загальні практики кодування

Суворе дотримання перевірених принципів (OWASP Secure Coding Practices) — використання актуальних, підтримуваних бібліотек, регулярне оновлення залежностей, проведення код-рев'ю з акцентом на безпеку й аудит — дозволяє виявити й усунути вразливості на ранніх етапах розробки та підтримувати стабільно високий рівень якості коду.

---

*Multipurpose Internet Mail Extensions.

2.3 Зіставлення OWASP Top 10 та OWASP Go Secure Coding Practices Guide

OWASP Top 10 та OWASP Go Secure Coding Practices Guide — це два взаємодоповнюючі ресурси, які разом формують потужну основу для забезпечення безпеки вебдодатків, що розробляються мовою Go. Зокрема, OWASP Top 10 визначає, що є найкритичнішими чинниками загроз у вебдодатках, тоді як OWASP Go Secure Coding Practices Guide дає настанови та приклади того, як можна вирішувати ці проблеми при розробці на Go.

Таблиця 2.2

Відповідність між OWASP Go Secure Coding Practices Guide та OWASP Top 10 2021

OWASP Top 10 (2021)	**Рекомендації з Go Secure Coding Practices Guide**
A01:2021 – Порушення контролю доступу	Контроль доступу: Впровадження перевірок доступу на стороні сервера, уникнення довіри до даних з клієнта.
A02:2021 – Криптографічні помилки	Криптографічні практики: Використання перевірених алгоритмів, належне управління ключами, уникнення застарілих методів.
A03:2021 – Ін’єкції	Валідація вхідних даних: Використання параметризованих запитів, уникнення динамічного SQL, ретельна перевірка введення.
A04:2021 – Небезпечний дизайн	Загальні практики кодування: Впровадження принципів безпеки на етапі

Продовження таблиці 2.2

OWASP Top 10 (2021)	**Рекомендації з Go Secure Coding Practices Guide**
	проектування, використання шаблонів безпечного дизайну.
A05:2021 – Неправильна конфігурація безпеки	Конфігурація системи: Забезпечення безпечних налаштувань за замовчуванням, обмеження доступу до адміністративних інтерфейсів.
A06:2021 – Вразливі та застарілі компоненти	Управління залежностями: Регулярне оновлення бібліотек, використання інструментів для виявлення вразливостей у залежностях.
A07:2021 – Помилки ідентифікації та автентифікації	Автентифікація та управління паролями: Використання надійних методів автентифікації, зберігання паролів з використанням хешування.
A08:2021 – Помилки цілісності програмного забезпечення та даних	Захист даних: Використання цифрових підписів, перевірка цілісності файлів, контроль джерел оновлень.
A09:2021 – Помилки журналювання та моніторингу	Обробка помилок та логування: Впровадження централізованого логування, моніторинг аномальної активності, захист логів від несанкціонованого доступу.
A10:2021 – SSRF (Підроблені запити з боку сервера)	Безпека комунікацій: Валідація URL-адрес, обмеження вихідних за-

Закінчення таблиці 2.2

OWASP Top 10 (2021)	**Рекомендації з Go Secure Coding Practices Guide**
	питів, використання списків дозволених адрес.

Таблиця 2.2 демонструє, як практичні рекомендації з безпечного програмування на Go допомагають вирішити ключові проблеми безпеки, визначені в OWASP Top 10 2021. Впровадження цих практик сприяє створенню більш захищених вебдодатків.

2.4 Механізми виявлення вразливостей

2.4.1 Основні методи виявлення вразливостей

Виявлення вразливостей у вебдодатках є ключовим етапом забезпечення їхньої безпеки. Існує два основних методи, які допомагають ідентифікувати слабкі місця в системах:

1. *Статичне тестування безпеки додатків* (SAST). Цей метод передбачає аналіз вихідного коду або байт-коду додатку без його виконання. SAST дозволяє виявити вразливості на ранніх етапах розробки, що сприяє зниженню витрат на їх усунення. Цей підхід є частиною «білого ящика», оскільки аналіз проводиться з повним доступом до коду. Інструменти статичного аналізу синтаксично досліджують текст програми, шукаючи у ньому фіксований набір шаблонів або правил. Теоретично, вони також можуть аналізувати скомпільовану форму програмного забезпечення. Цей підхід базується на інструментуванні коду, що дозволяє зіставляти скомпільовані компоненти з компонентами вихідного коду з метою виявлення проблем. Статичний аналіз може виконуватись вручну у вигляді рецензування або аудиту коду для різних цілей, зокрема для забезпечення безпеки, однак цей процес є трудомістким [23].

2. *Динамічне тестування безпеки додатків* (DAST). DAST здійснює аналіз працюючого додатку, взаємодіючи з ним як зовнішній користувач. Це дозволяє виявити вразливості, які проявляються під час реальної роботи системи, такі як SQL-ін'єкції або XSS. DAST є методом «чорного ящика», оскільки тестування проводиться без доступу до вихідного коду. Такі методи, як сканування вразливостей, тестування на проникнення та фаззінг [24], можна класифікувати як частину DAST. Цей метод є високомасштабованим, легко інтегрується та працює швидко. Інструменти динамічного аналізу безпеки (DAST) добре підходять для виявлення атак низького рівня, таких як ін'єкційні вразливості, але неефективні для виявлення вразливостей високого рівня, наприклад, помилок в логіці чи бізнес-логіці додатку [25].

Для того, щоб реалізувати приклад безпечного кодування вебдодатків, необхідно користуватися обома методами. Зважаючи на це, статичне тестування безпеки для моєї демонстрації було виконано вручну за допомогою вилучення фрагментів коду, що не відповідають шаблонам та правилам безпеки, які базуються на аналізі проєктів OWASP, згаданих вище, або заміни їх на ті, що таким вимогам відповідають та зберігають чи вводять бажаний функціонал, а динамічне дестування безпеки — за допомогою одного із засобів автоматизованого сканування вразливостей, які розглянуто нижче.

2.4.2 Сканери вебдодатків

Сканери вебдодатків здійснюють дослідження вебдодатку шляхом автоматизованого обходу його вебсторінок з метою виявлення вразливостей безпеки. Такий процес включає генерацію потенційно шкідливих вхідних даних і аналіз реакції додатку на ці дані, що дозволяє виявити недоліки в його обробці запитів [25]. Існує багато інструментів для автоматизованого сканування вразливостей, з яких можна навести такі найбільш поширені:

– ZAP — це потужний відкритий інструмент для динамічного аналізу безпеки вебдодатків. Він призначений як для новачків, так і для досвідчених

фахівців, забезпечуючи можливість перехоплення HTTP/HTTPS-трафіку, автоматичного сканування вразливостей, а також ручного тестування. ZAP активно підтримується спільнотою OWASP, має плагіни для розширення функціональності та дозволяє інтегруватися в CI/CD-процеси.

– Burp Suite — професійне рішення для комплексного тестування безпеки вебдодатків. Інструмент включає набір модулів, зокрема проксі-сервер для перехоплення трафіку, сканер вразливостей, інструмент для фаззінгу, модуль повторного надсилання запитів (repeater), а також засіб для автоматизованого пошуку вразливостей. Burp Suite широко використовується в пентестингу та оцінці безпеки вебресурсів.

– SQLMap — автоматизований інструмент для виявлення та експлуатації SQL-ін'єкцій у вебзастосунках. SQLMap підтримує численні СУБД (MySQL, PostgreSQL, Oracle, Microsoft SQL Server тощо), здатен виявляти різні типи SQL-ін'єкцій (включаючи time-based, boolean-based та error-based) і навіть дозволяє отримувати повний дамп БД або доступ до файлової системи сервера через вразливість.

– Nessus — потужний сканер вразливостей, що дозволяє здійснювати перевірку систем, мереж та вебдодатків на наявність широкого спектра проблем безпеки. Nessus має велику базу сигнатур та дозволяє виявляти вразливості операційних систем, мережевих сервісів, програмного забезпечення та конфігурацій. Хоча Nessus не спеціалізується виключно на вебзастосунках, він є незамінним у загальному процесі оцінювання безпеки IT-інфраструктури.

– Nikto — відкритий вебсканер, що здійснює перевірку вебсерверів на наявність потенційно небезпечних файлів, застарілих версій програмного забезпечення, помилок конфігурації та інших відомих вразливостей. Nikto не є «тихим» сканером і не призначений для стелс-аналізу, однак залишається ефективним засобом первинного аудиту безпеки вебресурсів.

Ці інструменти доповнюють один одного у процесі комплексного тестування безпеки вебдодатків і дозволяють виявляти як типові, так і складніші

вразливості, що можуть становити загрозу для конфіденційності, цілісності та доступності даних.

Висновки за розділом 2

В другому розділі кваліфікаційної роботи було розглянуто питання, що безпосередньо відносяться до аналізу та виявлення вразливостей вебдодатків. Можна зазначити, що безпека вебдодатків є ключовим елементом сучасного програмного забезпечення, особливо в умовах зростаючої складності систем і зловмисної активності. У цьому розділі було розглянуто широкий спектр типових вразливостей, які виникають у процесі розробки та експлуатації вебдодатків, а також практичні рекомендації щодо їх запобігання.

Огляд OWASP Top 10 2021 дав змогу співвіднести найбільш критичні вразливості з відповідними практиками безпечного кодування, зокрема у контексті мови Go. Це показує, що впровадження системного підходу до безпеки, включно з інструментами статичного аналізу, контрольованими залежностями та ретельною обробкою помилок, дозволяє значно зменшити ризики.

Використання рекомендацій з OWASP Go Secure Coding Practices Guide у поєднанні з аналізом OWASP Top 10 дозволяє виявити та ефективно усунути найпоширеніші вразливості, властиві вебдодаткам. Практичні заходи, такі як параметризовані запити, кодування вихідних даних, багатофакторна автентифікація, належне логування, а також регулярне оновлення компонентів, мають стати невід'ємною частиною повсякденної роботи розробників.

Особливо важливо не лише теоретично розуміти загрози, а й послідовно впроваджувати захисні механізми у власному коді, дотримуючись принципів «secure by design», але при цьому не нехтуючи виконанням статичного та динамічного тестування безпеки, що може бути здійснено за допомогою автоматизованих інструментів. Це дозволить суттєво підвищити загальний рівень безпеки інформаційних систем і зменшити ризики кіберінцидентів.

Розділ 3. ВПРОВАДЖЕННЯ МЕХАНІЗМІВ ЗАХИСТУ ВЕБДОДАТКІВ НА ОСНОВІ МОВИ ПРОГРАМУВАННЯ GO

3.1 Структура демонстраційного вебдодатку

3.1.1 Технологічний стек і залежності

Демонстраційний вебдодаток реалізовано на Go 1.23 із стандартними та зовнішніми бібліотеками, які забезпечують увесь необхідний функціонал безпеки та роботи з HTTP, шаблонами та БД. Основні залежності:

– стандартна бібліотека Go:

– `net/http` – реалізація HTTP-сервера й маршрутизації.

– `html/template` – безпечне відображення HTML із автоматичним екрануванням.

– `database/sql` – загальний інтерфейс до СУБД.

– `crypto/rand`, `encoding/base64` – криптографічно стійка генерація випадкових токенів.

– `regexp`, `strings`, `strconv`, `time` – допоміжні пакети для валідації, обробки рядків та часу.

– зовнішні пакети:

– `_ "github.com/mattn/go-sqlite3"` – драйвер SQLite (для демонстраційної in-memory БД).

– `golang.org/x/crypto/bcrypt` – адаптивне хешування паролів зі salt.

Для запобігання CSRF була продемонстрована власна реалізація токенів, тоді як опціонально можна використати пакет `gorilla/csrf`.

3.1.2 Основні компоненти

1. Структура App — кореневий об’єкт, що тримає з’єднання з БД (`*sql.DB`) та карту сесій (`map[string]Session`).

2. ППЗ* безпеки (`securityHeaders`) — встановлює низку HTTP-заголовків, що відповідають за протидію XSS, CSP з nonce тощо.

3. ППЗ автентифікації (`requireAuth`) — перевіряє дійсність та термін дії cookie “session_token”, оновлює час життя сесії.

4. Хендлери:

– `registerHandler` і `loginHandler` — обробка форм реєстрації й входу з CSRF-токенами у тимчасових cookie, валідація через регулярні вирази, bcrypt-хешування.

– `dashboardHandler` — захищений маршрут, який витягує дані користувача з БД за session.UserID.

– `apiUsersHandler` — REST-ендпоінт, що повертає JSON-список користувачів via `json.NewEncoder` з параметризованою SQL-вибіркою.

5. БД — in-memory SQLite, ініціалізована `CREATE TABLE users` у `initDB()`; усі запити виконуються через підготовлені вирази (`Prepare + Exec/Query`) для запобігання SQL-ін’єкціям.

Разом, цей набір компонентів демонструє принципи «secure by design»: чітка структура проекту, використання адаптованих алгоритмів криптографії, підхід з використанням ППЗ для накладення аспектів безпеки та інкапсуляція даних у моделі й шаблонах.

3.2 Реалізація механізмів захисту

3.2.1 Встановлення заголовків безпеки

У демонстраційному додатку всі HTTP-ендпоінти обгорнуті в ППЗ `securityHeaders`, яке встановлює набір заголовків безпеки:

– X-Content-Type-Options: nosniff, X-Frame-Options: DENY, X-XSS-Protection: 1; mode=block для запобігання XSS та clickjacking.

– Strict-Transport-Security (HSTS) для примусового HTTPS.

*Переклад терміну «middleware».

– Cross-Origin-Resource-Policy, Cross-Origin-Opener-Policy, Cross-Origin-Embedder-Policy, Permissions-Policy для мінімізації ризиків Spectre/Meltdown і обмеження доступу до API браузера.

– Content-Security-Policy зі згенерованим динамічним nonce, який подається в шаблони для inline-стилів і скриптів, а також обмежує джерела ресурсів до власного домену.

3.2.2 Обробка маршрутизації та захист від CSRF

Для маршрутизації використовується стандартний `http.HandleFunc`, а перед обробником виконується перевірка CSRF-токену:

– при GET-запиті `registerHandler` генерує випадковий CSRF-токен, зберігає його в тимчасовому `HttpOnly`-cookie та вставляє в HTML-форму.

– при POST-запиті цей токен порівнюється зі значенням cookie; у разі невідповідності запит відхиляється з кодом 403 Forbidden.

3.2.3 Фільтрація й нормалізація URL та параметрів

У всіх обробниках REST-запитів (наприклад, `/api/users`) значення з `r.URL.Query()` проходять первинну перевірку:

– параметри, що впливають на логіку (наприклад, `limit`), конвертуються через `strconv.Atoi` і ретельно перевіряються на діапазон (1–100).

– для побудови SQL-запитів заборонена конкатенація рядків — використовується виключно підготовлені вирази (`Prepare` + `Query`).

3.2.4 Реєстрація та хешування паролів

У `registerHandler` після валідації полів виконується адаптивне хешування за допомогою bcrypt з cost=12. Функція `hashPassword` викликає `bcrypt.GenerateFromPassword` і повертає збережений у базі текстовий рядок хешу.

3.2.5 Робота з cookie та токенами сесій

Після успішної авторизації в `loginHandler` генерується криптографічно стійкий session_token через `generateSessionToken`, зберігається в пам'яті сервера разом із терміном дії та CSRF-токеном, і відправляється клієнтові в `HttpOnly`, `Secure`, `SameSite=Strict` cookie з MaxAge 1800 секунд. Middleware `requireAuth` перевіряє наявність та дійсність цього cookie, оновлює `ExpiresAt` сесії й перенаправляє на сторінку входу у разі прострочення або відсутності сесії.

3.2.6 Серверна валідація вхідних полів

Перш ніж зберігати користувацькі дані, застосовується набір регулярних виразів:

– `validateEmail` перевіряє синтаксис email і довжину ≤ 254 символів.

– `validateUsername` дозволяє лише букви, цифри та підкреслення довжиною 3–30 символів.

– `validatePassword` контролює наявність великих/малих літер, цифр та спеціальних символів, а також довжину пароля (8–128).

3.2.7 Параметризовані SQL-запити для запобігання SQL-ін'єкціям

Усі робочі операції з базою даних виконуються за допомогою підготовлених запитів (див. лістинг 3.1).

```go
stmt, _ := app.db.Prepare("INSERT INTO users (username, email,
password_hash) VALUES (?, ?, ?)")
stmt.Exec(username, email, hashedPassword)
```

```go
stmt, _ := app.db.Prepare("SELECT id, username, email FROM
users LIMIT ?")
stmt.Query(limit)
```

Лістинг 3.1 — Підготовлені запити (prepared statements)

Це запобігає будь-якій ін’єкції, оскільки драйвер самостійно екранує вхідні параметри.

3.2.8 Кодування вихідних даних для захисту від XSS

Для відображення динамічного контенту використовується пакет `html/template`, який автоматично ескейпить всі змінні у HTML. У випадку вставки даних у JavaScript-контекст застосовується `template.JS` з JSON-серіалізацією, щоб уникнути розриву рядка або виконання довільного коду.

3.3 Тестування вебдодатку та сканування на вразливості

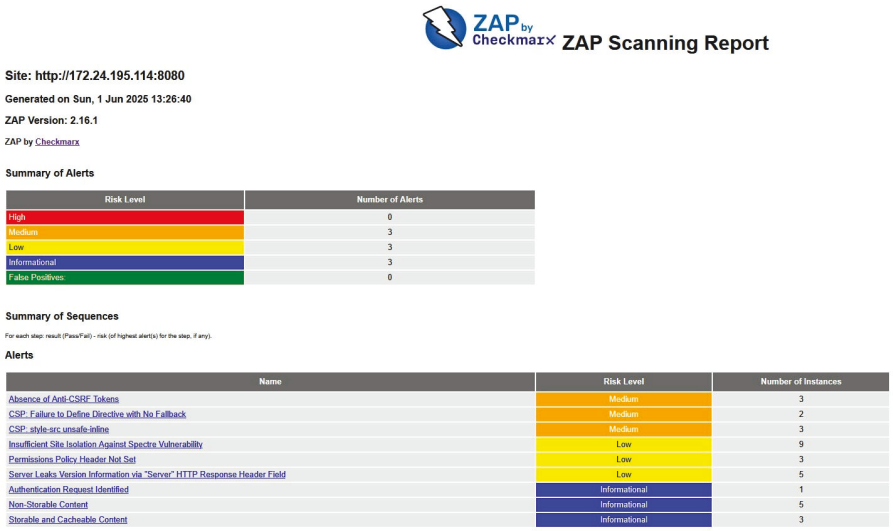
3.3.1 Автоматизоване сканування за допомогою ZAP

Для динамічного аналізу безпеки було використано Zed Attack Proxy by Checkmarx (ZAP). У Додатку Б представлено скрипт *scan.sh*, який запускає контейнер zaproxy/zap-stable та виконує базове сканування ендпоінтів додатку через `zap-baseline.py` (параметри `-t` і `-r` для цільового URL і імені звіту).

1. Налаштування середовища: скрипт визначає локальний IP-адрес інтерфейсу eth0 та порт 8080.

2. Запуск сканування: ZAP запускається в “baseline” режимі, що перевіряє усі загальні типи вразливостей (SQLi, XSS, CSRF тощо).

3. Звіт: у поточній теці генерується HTML-файл із результатами, названий `scan-report-<timestamp>.html` (рисунок 3.1).



ZAP by Checkmarx ZAP Scanning Report

Site: http://172.24.195.114:8080

Generated on Sun, 1 Jun 2025 13:26:40

ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	3
Low	3
Informational	3
False Positives:	0

Summary of Sequences

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
Absence of Anti-CSRF Tokens	Medium	3
CSP: Failure to Define Directive with No Fallback	Medium	2
CSP: style-src unsafe-inline	Medium	3
Insufficient Site Isolation Against Spectre Vulnerability	Low	9
Permissions Policy Header Not Set	Low	3
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	5
Authentication Request Identified	Informational	1
Non-Storable Content	Informational	5
Storable and Cacheable Content	Informational	3

Рисунок 3.1 — Скріншот першого звіту сканування ZAP

3.3.2 Аналіз результатів і впровадження виправлень

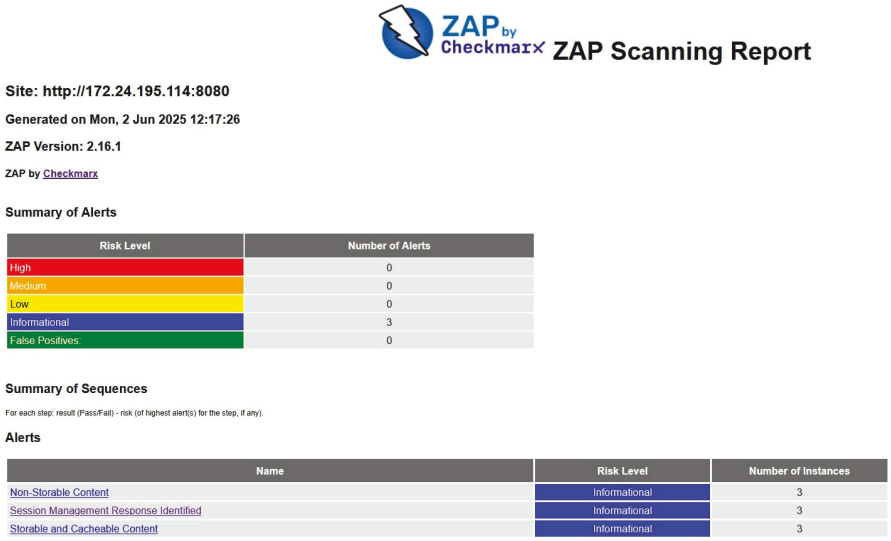
Після завершення сканування було проведено детальний огляд знайдених ZAP вразливостей:

– Відсутність окремих заголовків безпеки (HSTS, CSP з nonce, X-Frame-Options) було виявлено як “Security Misconfiguration”. Ці дефіцити було усунено шляхом доповнення middleware `securityHeaders`.

– Незахищені форми (відсутність CSRF-токена) позначилися як “Cross-Site Request Forgery” — додано генерацію й перевірку CSRF-токена в cookie та прихованих полях форм.

– Рекомендації щодо безпечного протоколу: ZAP наголосив на необхідності HSTS і TLS-Only налаштувань — у продакшн-режимі застосунок переведено на HTTPS із відповідними заголовками.

Кожна виявлена уразливість була задокументована, а в коді — виправлена та протестована повторним запуском скрипту. Усі критичні та помірні дефекти усунені, що підтверджує відсутність “High” і “Medium” ризиків у фінальному звіті сканування (рисунок 3.2).



ZAP by Checkmarx ZAP Scanning Report

Site: http://172.24.195.114:8080

Generated on Mon, 2 Jun 2025 12:17:26

ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	0
Low	0
Informational	3
False Positives:	0

Summary of Sequences

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
Non-Storable Content	Informational	3
Session Management Response Identified	Informational	3
Storable and Cacheable Content	Informational	3

Рисунок 3.2 — Скріншот фінального звіту сканування ZAP

Висновки за розділом 3

У результаті практичної частини кваліфікаційної роботи було успішно реалізовано демонстраційний вебдодаток мовою Go, основною метою якого стало впровадження механізмів безпечної розробки вебсистем. Структура додатку передбачає розділення на логічні компоненти, включаючи обробку HTTP-запитів, автентифікацію, управління сесіями, валідацію вхідних даних та логування.

Застосовано ключові практики OWASP Go Secure Coding Practices, такі як:

– обробка користувацького введення з валідацією на боці сервера;

– використання параметризованих SQL-запитів для запобігання SQL-ін’єкціям;

– реалізація автентифікації з використанням хешування паролів та токенів сесій;

– захист від XSS через правильне кодування виводу.

Крім того, з метою перевірки стійкості до типових вразливостей, вебдодаток було протестовано за допомогою інструмента динамічного аналізу ZAP, що дозволило виявити й усунути потенційні загрози ще на етапі розробки.

Весь програмний код демонстраційного додатку наведено в Додатку А.

Узагальнюючи, створений демонстраційний вебдодаток демонструє практичне застосування теоретичних знань з безпечного програмування, що були отримані завдяки аналізу мови програмування Go та дослідження вразливостей вебдодатків відповідно до OWASP Top 10 та OWASP Go Secure Coding Practices, та може слугувати основою для розробки реальних захищених вебсистем на основі мови програмування Go.

ВИСНОВКИ

У рамках кваліфікаційної роботи було досліджено проблематику забезпечення кібербезпеки вебдодатків на основі мови програмування Go.

У першому розділі кваліфікаційної роботи було теоретично проаналізовано особливості мови програмування Go, зокрема її синтаксис, типізацію та модульність, а також підхід до обробки помилок. Розглянуто інструменти, які формують екосистему Go, серед яких `go fmt`, `go vet`, `go test`, `go mod` тощо. Досліджено модель рівночасності на основі використання горутин і каналів. Окрему увагу приділено перевагам мови Go у розробці безпечних і масштабованих вебдодатків.

У другому розділі кваліфікаційної роботи було проаналізовано основні загрози безпеці вебдодатків згідно з OWASP Top 10 (2021), зокрема SQL-ін'єкції, XSS, порушення автентифікації, SSRF та інші поширені вразливості. Узагальнено рекомендації OWASP Go Secure Coding Practices для запобігання цим загрозам і проведено їх зіставлення з конкретними вразливостями у Go-додатках. Також розглянуто методи виявлення вразливостей та інструменти, що дозволяють перевіряти безпеку вебсистем.

У третьому розділі кваліфікаційної роботи було реалізовано фрагменти безпечного коду для обробки HTTP-запитів, валідації введення, автентифікації та логування. Поведінку вебдодатку протестовано на типові помилки безпеки. Розроблено та застосовано скрипт із використанням ZAP для виявлення вразливостей у розробленому додатку. На основі отриманих результатів створено демонстраційний вебдодаток, який реалізує принципи безпечного програмування на Go.

Створений у практичній частині вебдодаток включає в себе:

1. ППЗ (middleware) для встановлення захисних HTTP-заголовків (CSP, HSTS, X-Frame-Options тощо).

2. Механізми CSRF-захисту та безпечну маршрутизацію.

3. Адаптивне хешування паролів (bcrypt) та роботу з безпечними сесійними cookie.

4. Серверну валідацію даних із використанням регулярних виразів і зовнішніх пакетів.

5. Параметризовані SQL-запити й автоматичне кодування виводу через `html/template`.

Для підтвердження надійності застосовано динамічне тестування за допомогою ZAP, що дозволило виявити й усунути усі вразливості критичного, середнього та низького рівня.

Таким чином, продемонстрована реалізація не лише підтвердила ефективність рекомендованих практик, а й може служити шаблоном для побудови реальних високобезпечних вебсистем на Go. Незважаючи на досягнуті результати, до подальших робіт може належати інтеграція багатофакторної автентифікації, розгортання в контейнеризованому середовищі та розширене автоматизоване тестування (SAST, DAST).

Отже, можна стверджувати, що у ході виконання кваліфікаційної роботи виконано поставлені задачі, а саме було:

– проаналізовано особливості мови програмування Go в розрізі безпечної розробки вебдодатків;

– досліджено типові загрози безпеці вебдодатків згідно з OWASP Top 10;

– проаналізовано рекомендації OWASP Go Secure Coding Practices та їх відповідність до загроз OWASP;

– реалізовано приклад безпечного кодування вебдодатків на основі мови програмування Go з використанням інструментів для автоматизованого виявлення вразливостей.

Таким чином, в ході виконання кваліфікаційної роботи було досягнуто поставленої мети — розроблено практичні підходи до підвищення безпеки вебдодатків на базі мови Go з урахуванням рекомендацій OWASP.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Pike R. Go at Google: Language Design in the Service of Software Engineering. 2012. URL: https://go.dev/talks/2012/splash.article (дата звернення: 15.05.2025).
2. The Go Programming Language - TIOBE Index. 2025. URL: https://www.tiobe.com/tiobe-index/go/ (дата звернення: 15.05.2025).
3. Pike R. Go: Ten Years and Climbing. 2017. URL: https://commandcenter.blogspot.com/2017/09/go-ten-years-and-climbing.html (дата звернення: 15.05.2025).
4. The Go Authors. Frequently Asked Questions (FAQ). 2025. URL: https://go.dev/doc/faq (дата звернення: 15.05.2025).
5. Donovan A.A.A., Kernighan B.W. The Go Programming Language. Boston, MA: Addison-Wesley Professional, 2015.
6. The Go Authors. Go Language Design FAQ (Архівна версія). 2010. URL: https://web.archive.org/web/20100818220640/http://golang.org/doc/go_lang_faq.html (дата звернення: 15.05.2025).
7. Gerrand A. Go: One Year Ago Today. 2010. URL: https://blog.golang.org/2010/11/go-one-year-ago-today.html (дата звернення: 15.05.2025).
8. Gerrand A. Go version 1 is released. 2012. URL: https://go.dev/blog/go1 (дата звернення: 15.05.2025).
9. Cox R. Go Release Cycle. 2023. URL: https://github.com/golang/go/wiki/Go-Release-Cycle/20aaa9e3e86000c77a11735358e0d2375477c81e (дата звернення: 15.05.2025).
10. The Go Authors. Go 1.11 Release Notes. 2018. URL: https://go.dev/doc/go1.11 (дата звернення: 15.05.2025).
11. The Go Authors. Go 1.18 Release Notes. 2022. URL: https://go.dev/doc/go1.18 (дата звернення: 15.05.2025).
12. The Go Authors. Release History. 2025. URL: https://go.dev/doc/devel/release (дата звернення: 15.05.2025).

13. The Go Authors. The Go Programming Language Specification. 2024. URL: https://go.dev/ref/spec (дата звернення: 15.05.2025).
14. Kuhn M. UTF-8 history. 2002. URL: https://web.archive.org/web/20110301051538/http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt (дата звернення: 15.05.2025).
15. Pike R. Go Declaration Syntax. 2010. URL: https://go.dev/blog/declaration-syntax (дата звернення: 15.05.2025).
16. The Go Authors. Go Modules Reference. 2025. URL: https://go.dev/ref/mod (дата звернення: 15.05.2025).
17. The Go Authors. cmd/go - Go command documentation. 2025. URL: https://pkg.go.dev/cmd/go (дата звернення: 15.05.2025).
18. The Go Authors. Effective Go. 2022. URL: https://go.dev/doc/effective_go (дата звернення: 15.05.2025).
19. Вікіпедія. Вебзастосунок — Вікіпедія. 2025. URL: https://uk.wikipedia.org/w/index.php?title=%D0%92%D0%B5%D0%B1%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA&oldid=44473137 (дата звернення: 15.05.2025).
20. OWASP Foundation. OWASP Top Ten Web Application Security Risks. 2024. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 15.05.2025).
21. Wikipedia contributors. OWASP — Wikipedia, The Free Encyclopedia. 2025. URL: https://en.wikipedia.org/w/index.php?title=OWASP&oldid=1275004016 (дата звернення: 15.05.2025).
22. OWASP Foundation. OWASP Go Secure Coding Practices Guide. 2017. URL: https://owasp.org/www-project-go-secure-coding-practices-guide/ (дата звернення: 15.05.2025).
23. Chess B., McGraw G. Static analysis for security // IEEE Security & Privacy. 2004. вип. 2, № 6. с. 76–79.
24. OWASP Foundation. Fuzzing - OWASP. 2024. URL: https://owasp.org/www-community/Fuzzing (дата звернення: 15.05.2025).

25. National Institute of Standards and Technology. Web Application Scanners. 2024. URL: https://www.nist.gov/itl/ssd/software-quality-group/web-application-scanners (дата звернення: 15.05.2025).

Додаток А

ПРОГРАМНИЙ КОД ДОДАТКУ

main.go

```
package main

import (
  "crypto/rand"
  "database/sql"
  "encoding/base64"
  "encoding/json"
  "fmt"
  "html/template"
  "log"
  "net/http"
  "regexp"
  "strconv"
  "strings"
  "time"

  _ "github.com/mattn/go-sqlite3"
  "golang.org/x/crypto/bcrypt"
)

// User represents a user in the system
type User struct {
  ID       int    `json:"id"`
  Username string `json:"username"`
  Email    string `json:"email"`
  Password string `json:"-"` // Never expose password in JSON
}

// Session represents a user session
type Session struct {
  Token     string
  UserID    int
  ExpiresAt time.Time
  CSRFToken string // Add CSRF token to session
}

// App holds our application dependencies
type App struct {
  db       *sql.DB
  sessions map[string]Session
}

// Generate CSRF token
func generateCSRFToken() (string, error) {
  bytes := make([]byte, 32)
  if _, err := rand.Read(bytes); err != nil {
    return "", err
  }
  return base64.URLEncoding.EncodeToString(bytes), nil
}

// Security middleware for setting secure headers (OWASP A05: Security Misconfiguration)
func securityHeaders(next http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    // Prevent XSS attacks
    w.Header().Set("X-Content-Type-Options", "nosniff")
    w.Header().Set("X-Frame-Options", "DENY")
    w.Header().Set("X-XSS-Protection", "1; mode=block")

    // HTTPS enforcement (in production, use HSTS)
            w.Header().Set("Strict-Transport-Security",   "max-age=63072000; includeSubDomains; preload")

    // Cross-Origin-Resource-Policy for Spectre vulnerability protection
    // Set to 'same-origin' to prevent cross-origin access to resources
    w.Header().Set("Cross-Origin-Resource-Policy", "same-origin")

    // Cross-Origin-Opener-Policy for additional isolation
    w.Header().Set("Cross-Origin-Opener-Policy", "same-origin")

    // Cross-Origin-Embedder-Policy for enhanced security
    w.Header().Set("Cross-Origin-Embedder-Policy", "require-corp")

    // Permissions Policy for controlling browser features and APIs
    // Restrict potentially dangerous features to prevent misuse
    permissionsPolicy := "geolocation=(), " +
      "microphone=(), " +
      "camera=(), " +
      "payment=(), " +
      "usb=(), " +
      "accelerometer=(), " +
      "autoplay=(), " +
      "encrypted-media=(), " +
      "fullscreen=(self), " +
      "gyroscope=(), " +
      "magnetometer=(), " +
      "midi=(), " +
      "notifications=(), " +
      "push=(), " +
      "speaker-selection=(), " +
      "sync-xhr=(), " +
      "vibrate=(), " +
      "vr=(), " +
      "wake-lock=(), " +
      "web-share=(), " +
      "xr-spatial-tracking=()"
    w.Header().Set("Permissions-Policy", permissionsPolicy)

    // Enhanced Content Security Policy with nonce for styles
    nonce := generateNonce()
    csp := fmt.Sprintf(
      "default-src 'self'; "+
        "script-src 'self'; "+
        "style-src 'self' 'nonce-%s'; "+
        "img-src 'self' data:; "+
        "font-src 'self'; "+
        "connect-src 'self'; "+
        "media-src 'none'; "+
        "object-src 'none'; "+
        "child-src 'none'; "+
        "frame-src 'none'; "+
        "worker-src 'none'; "+
        "frame-ancestors 'none'; "+
        "form-action 'self'; "+
        "base-uri 'self'",
      nonce)
    w.Header().Set("Content-Security-Policy", csp)

    // Store nonce for template use
    r.Header.Set("X-CSP-Nonce", nonce)

    // Prevent information disclosure
    // w.Header().Set("Server", "SecureApp/1.0")

    next(w, r)
  }
}

// Generate nonce for CSP
func generateNonce() string {
  bytes := make([]byte, 16)
  rand.Read(bytes)
  return base64.StdEncoding.EncodeToString(bytes)
}

// Input validation functions (OWASP A03: Injection)
func validateEmail(email string) bool {
  emailRegex := regexp.MustCompile(`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`)
  return emailRegex.MatchString(email) && len(email) <= 254
}

func validateUsername(username string) bool {
  // Alphanumeric and underscore only, 3-30 characters
  usernameRegex := regexp.MustCompile(`^[a-zA-Z0-9_]{3,30}$`)
  return usernameRegex.MatchString(username)
}

func validatePassword(password string) bool {
  // At least 8 characters, with uppercase, lowercase, number, and special char
  if len(password) < 8 || len(password) > 128 {
    return false
  }

  hasUpper := regexp.MustCompile(`[A-Z]`).MatchString(password)
  hasLower := regexp.MustCompile(`[a-z]`).MatchString(password)
  hasNumber := regexp.MustCompile(`\d`).MatchString(password)
              hasSpecial       :=       regexp.MustCompile(`[!@#$%^&*(),.?":{}|<>]`).MatchString(password)

  return hasUpper && hasLower && hasNumber && hasSpecial
}

// Secure password hashing (OWASP A07: Identification and Authentication Failures)
func hashPassword(password string) (string, error) {
  // Use bcrypt with cost 12 (recommended minimum)
  hash, err := bcrypt.GenerateFromPassword([]byte(password), 12)
  return string(hash), err
}

func verifyPassword(password, hash string) bool {
  err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
  return err == nil
}

// Generate cryptographically secure session token
func generateSessionToken() (string, error) {
  bytes := make([]byte, 32)
  if _, err := rand.Read(bytes); err != nil {
    return "", err
  }
  return base64.URLEncoding.EncodeToString(bytes), nil
}

// Database initialization with prepared statements (OWASP A03: Injection)
func (app *App) initDB() error {
  var err error
```

```
  app.db, err = sql.Open("sqlite3", ":memory:")
  if err != nil {
    return err
  }

  // Create users table
  createUserTable := `
  CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE NOT NULL,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
  );`

  _, err = app.db.Exec(createUserTable)
  return err
}

// Authentication middleware (OWASP A07: Identification and Authentication Failures)
func (app *App) requireAuth(next http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("session_token")
    if err != nil {
      http.Redirect(w, r, "/login", http.StatusUnauthorized)
      return
    }

    session, exists := app.sessions[cookie.Value]
    if !exists || time.Now().After(session.ExpiresAt) {
      // Clean up expired session
      delete(app.sessions, cookie.Value)
      http.Redirect(w, r, "/login", http.StatusUnauthorized)
      return
    }

    // Extend session
    session.ExpiresAt = time.Now().Add(30 * time.Minute)
    app.sessions[cookie.Value] = session

    next(w, r)
  }
}

// Registration handler with input validation
func (app *App) registerHandler(w http.ResponseWriter, r *http.Request) {
  if r.Method == "GET" {
    // Generate CSRF token for the form
    csrfToken, _ := generateCSRFToken()
    nonce := r.Header.Get("X-CSP-Nonce")

    data := map[string]string{
      "CSRFToken": csrfToken,
      "Nonce":     nonce,
    }

    // Store CSRF token temporarily (in production, use a more robust storage)
    // For this example, we'll use a simple approach
    http.SetCookie(w, &http.Cookie{
      Name:     "csrf_temp",
      Value:    csrfToken,
      Path:     "/",
      HttpOnly: true,
      Secure:   true,
      SameSite: http.SameSiteStrictMode,
      MaxAge:   300, // 5 minutes
    })

    t, _ := template.ParseFiles("register.html")
    t.Execute(w, data)
    return
  }

  // Validate CSRF token for registration
  formCSRF := r.FormValue("csrf_token")
  tempCookie, err := r.Cookie("csrf_temp")
  if err != nil || formCSRF == "" || formCSRF != tempCookie.Value {
    http.Error(w, "Invalid CSRF token", http.StatusForbidden)
    return
  }

  // Clear temporary CSRF cookie
  http.SetCookie(w, &http.Cookie{
    Name:     "csrf_temp",
    Value:    "",
    Path:     "/",
    HttpOnly: true,
    MaxAge:   -1,
  })

  // Parse and validate input
  username := strings.TrimSpace(r.FormValue("username"))
  email := strings.TrimSpace(r.FormValue("email"))
  password := r.FormValue("password")

  // Server-side validation (never trust client-side validation)
  if !validateUsername(username) {
    http.Error(w, "Invalid username format", http.StatusBadRequest)
    return
  }

  if !validateEmail(email) {
    http.Error(w, "Invalid email format", http.StatusBadRequest)
    return
  }

  if !validatePassword(password) {
   http.Error(w, "Password does not meet requirements", http.StatusBadRequest)
    return
  }

  // Hash password
  hashedPassword, err := hashPassword(password)
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }

  // Use prepared statement to prevent SQL injection (OWASP A03: Injection)
    stmt, err := app.db.Prepare("INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)")
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }
  defer stmt.Close()

  _, err = stmt.Exec(username, email, hashedPassword)
  if err != nil {
    if strings.Contains(err.Error(), "UNIQUE constraint failed") {
      http.Error(w, "Username or email already exists", http.StatusConflict)
      return
    }
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }

  http.Redirect(w, r, "/login?registered=true", http.StatusSeeOther)
}

// Login handler with rate limiting protection
func (app *App) loginHandler(w http.ResponseWriter, r *http.Request) {
  if r.Method == "GET" {
    // Generate CSRF token for the form
    csrfToken, _ := generateCSRFToken()
    nonce := r.Header.Get("X-CSP-Nonce")

    message := ""
    if r.URL.Query().Get("registered") == "true" {
      message = "Registration successful! Please login."
    }

    // Store CSRF token temporarily
    http.SetCookie(w, &http.Cookie{
      Name:     "csrf_temp",
      Value:    csrfToken,
      Path:     "/",
      HttpOnly: true,
      Secure:   true,
      SameSite: http.SameSiteStrictMode,
      MaxAge:   300, // 5 minutes
    })

    data := map[string]string{
      "Message":   message,
      "CSRFToken": csrfToken,
      "Nonce":     nonce,
    }

    t, _ := template.ParseFiles("login.html")
    t.Execute(w, data)
    return
  }

  // Validate CSRF token for login
  formCSRF := r.FormValue("csrf_token")
  tempCookie, err := r.Cookie("csrf_temp")
  if err != nil || formCSRF == "" || formCSRF != tempCookie.Value {
    http.Error(w, "Invalid CSRF token", http.StatusForbidden)
    return
  }

  // Clear temporary CSRF cookie
  http.SetCookie(w, &http.Cookie{
    Name:     "csrf_temp",
    Value:    "",
    Path:     "/",
    HttpOnly: true,
    MaxAge:   -1,
  })

  username := strings.TrimSpace(r.FormValue("username"))
  password := r.FormValue("password")

  // Retrieve user from database using prepared statement
   stmt, err := app.db.Prepare("SELECT id, password_hash FROM users WHERE username = ?")
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }
  defer stmt.Close()

  var userID int
  var storedHash string
  err = stmt.QueryRow(username).Scan(&userID, &storedHash)
  if err != nil {
    http.Error(w, "Invalid credentials", http.StatusUnauthorized)
    return
  }

  // Verify password
  if !verifyPassword(password, storedHash) {
    http.Error(w, "Invalid credentials", http.StatusUnauthorized)
    return
  }

  // Create session with CSRF token
  sessionToken, err := generateSessionToken()
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
```

```
    return
  }

  csrfToken, err := generateCSRFToken()
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }

  session := Session{
    Token:     sessionToken,
    UserID:    userID,
    ExpiresAt: time.Now().Add(30 * time.Minute),
    CSRFToken: csrfToken,
  }

  app.sessions[sessionToken] = session

  // Set secure cookie (OWASP A07: Identification and Authentication Failures)
  cookie := &http.Cookie{
    Name:     "session_token",
    Value:    sessionToken,
    Path:     "/",
    HttpOnly: true, // Prevent XSS access to cookie
    Secure:   true, // HTTPS only (set to false for local development)
    SameSite: http.SameSiteStrictMode,
    MaxAge:   1800, // 30 minutes
  }
  http.SetCookie(w, cookie)

  http.Redirect(w, r, "/dashboard", http.StatusSeeOther)
}

// Dashboard handler (protected route)
func (app *App) dashboardHandler(w http.ResponseWriter, r *http.Request) {
  cookie, _ := r.Cookie("session_token")
  session := app.sessions[cookie.Value]

  // Get user info
  stmt, err := app.db.Prepare("SELECT username, email FROM users WHERE id = ?")
  if err != nil {
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }
  defer stmt.Close()

  var username, email string
  err = stmt.QueryRow(session.UserID).Scan(&username, &email)
  if err != nil {
    http.Error(w, "User not found", http.StatusNotFound)
    return
  }

  nonce := r.Header.Get("X-CSP-Nonce")

  t, _ := template.ParseFiles("dashboard.html")
  data := map[string]any{
    "Username":      username,
    "Email":         email,
    "ID":            session.UserID,
    "SessionExpiry": session.ExpiresAt.Format("2006-01-02 15:04:05"),
    "CSRFToken":     session.CSRFToken,
    "Nonce":         nonce,
  }
  t.Execute(w, data)
}

// Logout handler
func (app *App) logoutHandler(w http.ResponseWriter, r *http.Request) {
  cookie, err := r.Cookie("session_token")
  if err == nil {
    // Remove session from memory
    delete(app.sessions, cookie.Value)

    // Clear cookie
    http.SetCookie(w, &http.Cookie{
      Name:     "session_token",
      Value:    "",
      Path:     "/",
      HttpOnly: true,
      MaxAge:   -1,
    })
  }

  http.Redirect(w, r, "/login", http.StatusSeeOther)
}

// API endpoint with proper error handling (OWASP A09: Security Logging and Monitoring Failures)
func (app *App) apiUsersHandler(w http.ResponseWriter, r *http.Request) {
  // Only allow GET requests
  if r.Method != "GET" {
    w.Header().Set("Allow", "GET")
    http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
    return
  }

  // Parse query parameters safely
  limitStr := r.URL.Query().Get("limit")
  limit := 10 // default
  if limitStr != "" {
    if l, err := strconv.Atoi(limitStr); err == nil && l > 0 && l <= 100 {
      limit = l
    }
  }

  // Use prepared statement
  stmt, err := app.db.Prepare("SELECT id, username, email FROM users LIMIT ?")
  if err != nil {
    log.Printf("Database error: %v", err)
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }
  defer stmt.Close()

  rows, err := stmt.Query(limit)
  if err != nil {
    log.Printf("Query error: %v", err)
    http.Error(w, "Internal server error", http.StatusInternalServerError)
    return
  }
  defer rows.Close()

  var users []User
  for rows.Next() {
    var user User
    err := rows.Scan(&user.ID, &user.Username, &user.Email)
    if err != nil {
      log.Printf("Scan error: %v", err)
      continue
    }
    users = append(users, user)
  }

  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(users)
}

func main() {
  app := &App{
    sessions: make(map[string]Session),
  }

  // Initialize database
  if err := app.initDB(); err != nil {
    log.Fatal("Failed to initialize database:", err)
  }
  defer app.db.Close()

  // Routes with security middleware and CSRF protection
  http.HandleFunc("/register", securityHeaders(app.registerHandler))
  http.HandleFunc("/login", securityHeaders(app.loginHandler))
  http.HandleFunc("/logout", securityHeaders(app.logoutHandler))
                                                          http.HandleFunc("/dashboard", securityHeaders(app.requireAuth(app.dashboardHandler)))
                                                        http.HandleFunc("/api/users", securityHeaders(app.requireAuth(app.apiUsersHandler)))

  // Root redirect
  http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    http.Redirect(w, r, "/login", http.StatusSeeOther)
  })

  fmt.Println("🔒 Secure Go Web App starting on :8080")
  fmt.Println("🛡️ OWASP 2021 Top 10 protections implemented:")
  fmt.Println("   A01: Broken Access Control - Session-based authorization")
  fmt.Println("   A02: Cryptographic Failures - bcrypt hashing, secure tokens")
  fmt.Println("   A03: Injection - SQL injection prevention with prepared statements")
  fmt.Println("   A04: Insecure Design - Secure authentication flow design")
  fmt.Println("   A05: Security Misconfiguration - Security headers implemented")
  fmt.Println("   A06: Vulnerable Components - Using maintained crypto libraries")
  fmt.Println("   A07: Identity & Auth Failures - Strong passwords, secure sessions")
  fmt.Println("   A08: Software & Data Integrity - Input validation implemented")
  fmt.Println("   A09: Security Logging Failures - Error logging implemented")
  fmt.Println("   A10: Server-Side Request Forgery - Input validation on parameters")
  fmt.Println("🛡️ Additional protections:")
  fmt.Println("   - CSRF Protection with tokens")
  fmt.Println("   - Enhanced CSP with nonces and explicit directives")
  fmt.Println("   - Removed unsafe-inline from style-src")

  log.Fatal(http.ListenAndServe(":8080", nil))
}
```

go.mod

```
module secure-webapp

go 1.23.0

toolchain go1.23.9

require (
  github.com/mattn/go-sqlite3 v1.14.28
  golang.org/x/crypto v0.38.0
)
```

go.sum

```
github.com/mattn/go-sqlite3      v1.14.28      h1:ThEiQrnbtumT+QMknw63Befp/ce/nUPgBPMlRFEum7A=
github.com/mattn/go-sqlite3                                                v1.14.28/go.mod h1:Uh1q+B4BYcTPb+yiD3kU8Ct7aC0hY9fxUwlHK0RXw+Y=
golang.org/x/crypto v0.38.0 h1:jt+WWG8IZlBnVbomuhg2Mdq0+BBQaHbtqHEFEigjUV8=
golang.org/x/crypto                                                                                v0.38.0/go.mod h1:MvrbAqul58NNYPKnOra203SB9vpuZW0e+RRZV+Ggqjw=
```

register.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Secure Registration</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <style nonce="{{.Nonce}}">
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
            display: flex;
            align-items: center;
            justify-content: center;
            padding: 20px;
        }

        .container {
            background: white;
            padding: 40px;
            border-radius: 16px;
            box-shadow: 0 20px 40px rgba(0,0,0,0.1);
            width: 100%;
            max-width: 400px;
            backdrop-filter: blur(10px);
        }

        h2 {
            text-align: center;
            color: #333;
            margin-bottom: 30px;
            font-size: 28px;
            font-weight: 600;
        }

        .form-group {
            margin-bottom: 20px;
        }

        label {
            display: block;
            margin-bottom: 8px;
            color: #555;
            font-weight: 500;
        }

        input {
            width: 100%;
            padding: 12px 16px;
            border: 2px solid #e1e5e9;
            border-radius: 8px;
            font-size: 16px;
            transition: all 0.3s ease;
            background: #f8f9fa;
        }

        input:focus {
            outline: none;
            border-color: #667eea;
            background: white;
            box-shadow: 0 0 0 3px rgba(102, 126, 234, 0.1);
        }

        button {
            width: 100%;
            padding: 14px;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            color: white;
            border: none;
            border-radius: 8px;
            font-size: 16px;
            font-weight: 600;
            cursor: pointer;
            transition: all 0.3s ease;
            margin-top: 10px;
        }

        button:hover {
            transform: translateY(-2px);
            box-shadow: 0 10px 20px rgba(102, 126, 234, 0.3);
        }

        .link {
            text-align: center;
            margin-top: 20px;
        }

        .link a {
            color: #667eea;
            text-decoration: none;
            font-weight: 500;
        }

        .link a:hover {
            text-decoration: underline;
        }

        small {
            color: #666;
            font-size: 12px;
            margin-top: 5px;
            display: block;
        }

        .security-badge {
            background: #e8f5e8;
            color: #2d5a2d;
            padding: 8px 12px;
            border-radius: 6px;
            font-size: 12px;
            text-align: center;
            margin-bottom: 20px;
            border: 1px solid #c3e6c3;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="security-badge">🔒 Secure Registration</div>
        <h2>Create Account</h2>
        <form method="POST" action="/register">
            <input type="hidden" name="csrf_token" value="{{.CSRFToken}}">
            <div class="form-group">
                <label>Username</label>
                <input type="text" name="username" required maxlength="30" pattern="[a-zA-Z0-9_]{3,30}" placeholder="Enter your username">
            </div>
            <div class="form-group">
                <label>Email</label>
                <input type="email" name="email" required maxlength="254" placeholder="Enter your email">
            </div>
            <div class="form-group">
                <label>Password</label>
                <input type="password" name="password" required minlength="8" maxlength="128" placeholder="Create a strong password">
                <small>Must include uppercase, lowercase, number, and special character</small>
            </div>
            <button type="submit">Create Account</button>
        </form>
        <div class="link">
            <a href="/login">Already have an account? Sign in</a>
        </div>
    </div>
</body>
</html>
```

login.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Secure Login</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <style nonce="{{.Nonce}}">
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
            display: flex;
            align-items: center;
            justify-content: center;
            padding: 20px;
        }

        .container {
            background: white;
            padding: 40px;
            border-radius: 16px;
            box-shadow: 0 20px 40px rgba(0,0,0,0.1);
            width: 100%;
            max-width: 400px;
            backdrop-filter: blur(10px);
        }

        h2 {
            text-align: center;
            color: #333;
            margin-bottom: 30px;
            font-size: 28px;
            font-weight: 600;
        }

        .form-group {
            margin-bottom: 20px;
        }

        label {
            display: block;
            margin-bottom: 8px;
            color: #555;
            font-weight: 500;
        }

        input {
            width: 100%;
            padding: 12px 16px;
```

```
            border: 2px solid #e1e5e9;
            border-radius: 8px;
            font-size: 16px;
            transition: all 0.3s ease;
            background: #f8f9fa;
        }

        input:focus {
            outline: none;
            border-color: #667eea;
            background: white;
            box-shadow: 0 0 0 3px rgba(102, 126, 234, 0.1);
        }

        button {
            width: 100%;
            padding: 14px;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            color: white;
            border: none;
            border-radius: 8px;
            font-size: 16px;
            font-weight: 600;
            cursor: pointer;
            transition: all 0.3s ease;
            margin-top: 10px;
        }

        button:hover {
            transform: translateY(-2px);
            box-shadow: 0 10px 20px rgba(102, 126, 234, 0.3);
        }

        .link {
            text-align: center;
            margin-top: 20px;
        }

        .link a {
            color: #667eea;
            text-decoration: none;
            font-weight: 500;
        }

        .link a:hover {
            text-decoration: underline;
        }

        .success-message {
            background: #e8f5e8;
            color: #2d5a2d;
            padding: 12px;
            border-radius: 8px;
            margin-bottom: 20px;
            text-align: center;
            border: 1px solid #c3e6c3;
        }

        .security-badge {
            background: #e8f5e8;
            color: #2d5a2d;
            padding: 8px 12px;
            border-radius: 6px;
            font-size: 12px;
            text-align: center;
            margin-bottom: 20px;
            border: 1px solid #c3e6c3;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="security-badge">🔒 Secure Login</div>
        <h2>Welcome Back</h2>
        {{if .Message}}<div class="success-message">{{.Message}}</div>{{end}}
        <form method="POST" action="/login">
            <input type="hidden" name="csrf_token" value="{{.CSRFToken}}">
            <div class="form-group">
                <label>Username</label>
                <input type="text" name="username" required placeholder="Enter
your username">
            </div>
            <div class="form-group">
                <label>Password</label>
                <input type="password" name="password" required placeholder="Enter
your password">
            </div>
            <button type="submit">Sign In</button>
        </form>
        <div class="link">
            <a href="/register">Need an account? Create one</a>
        </div>
    </div>
</body>
</html>
```

dashboard.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Secure Dashboard</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <style nonce="{{.Nonce}}">
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
            padding: 20px;
        }

        .container {
            max-width: 900px;
            margin: 0 auto;
            background: white;
            border-radius: 16px;
            box-shadow: 0 20px 40px rgba(0,0,0,0.1);
            overflow: hidden;
        }

        .header {
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            color: white;
            padding: 30px 40px;
            text-align: center;
        }

        .header h2 {
            font-size: 32px;
            margin-bottom: 10px;
            font-weight: 600;
        }

        .content {
            padding: 40px;
        }

        .user-info {
            background: #f8f9fa;
            padding: 25px;
            border-radius: 12px;
            margin-bottom: 30px;
            border-left: 4px solid #667eea;
        }

        .user-info h3 {
            color: #333;
            margin-bottom: 15px;
            font-size: 20px;
        }

        .info-grid {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
            gap: 15px;
        }

        .info-item {
            display: flex;
            align-items: center;
            gap: 10px;
        }

        .info-label {
            font-weight: 600;
            color: #555;
        }

        .info-value {
            color: #333;
        }

        .security-section {
            margin-top: 30px;
        }

        .security-section h3 {
            color: #333;
            margin-bottom: 20px;
            font-size: 22px;
            display: flex;
            align-items: center;
            gap: 10px;
        }

        .security-grid {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
            gap: 15px;
        }

        .security-item {
            background: #e8f5e8;
            padding: 15px 20px;
            border-radius: 8px;
            display: flex;
            align-items: center;
            gap: 12px;
            border: 1px solid #c3e6c3;
            transition: all 0.3s ease;
        }

        .security-item:hover {
            transform: translateY(-2px);
            box-shadow: 0 5px 15px rgba(0,0,0,0.1);
        }

        .check-icon {
            color: #2d5a2d;
            font-weight: bold;
            font-size: 18px;
        }
```

```
        .security-text {
            color: #2d5a2d;
            font-weight: 500;
        }

        .owasp-code {
            font-family: 'Courier New', monospace;
            font-weight: bold;
            color: #764ba2;
        }

        .logout-section {
            margin-top: 40px;
            text-align: center;
            padding-top: 30px;
            border-top: 2px solid #e1e5e9;
        }

        .logout-btn {
            display: inline-block;
            padding: 12px 30px;
            background: linear-gradient(135deg, #dc3545 0%, #c82333 100%);
            color: white;
            text-decoration: none;
            border-radius: 8px;
            font-weight: 600;
            transition: all 0.3s ease;
        }

        .logout-btn:hover {
            transform: translateY(-2px);
            box-shadow: 0 8px 20px rgba(220, 53, 69, 0.3);
        }

        .session-info {
            background: #fff3cd;
            border: 1px solid #ffeaa7;
            color: #856404;
            padding: 12px 20px;
            border-radius: 8px;
            margin-top: 20px;
            text-align: center;
        }

        @media (max-width: 768px) {
            .info-grid {
                grid-template-columns: 1fr;
            }

            .security-grid {
                grid-template-columns: 1fr;
            }

            .content {
                padding: 20px;
            }

            .header {
                padding: 20px;
            }
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="header">
            <h2>🛡️ Secure Dashboard</h2>
            <p>OWASP 2021 Compliant Web Application</p>
        </div>

        <div class="content">
            <div class="user-info">
                <h3>👤 User Information</h3>
                <div class="info-grid">
                    <div class="info-item">
                        <span class="info-label">Username:</span>
                        <span class="info-value">{{.Username}}</span>
                    </div>
                    <div class="info-item">
                        <span class="info-label">Email:</span>
                        <span class="info-value">{{.Email}}</span>
                    </div>
                    <div class="info-item">
                        <span class="info-label">User ID:</span>
                        <span class="info-value">#{{.ID}}</span>
                    </div>
                </div>
                <div class="session-info">
                    ⏰ Session expires: {{.SessionExpiry}}
                </div>
            </div>

            <div class="security-section">
                <h3>🔒 OWASP 2021 Top 10 Security Features Implemented</h3>
                <div class="security-grid">
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A01</span>: Broken Access Control - Session-based authorization middleware</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A02</span>: Cryptographic Failures - bcrypt password hashing, secure session tokens</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A03</span>: Injection - SQL injection prevention via prepared statements</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A04</span>: Insecure Design - Secure authentication flow and session management</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A05</span>: Security Misconfiguration - Security headers and CSP implemented</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A06</span>: Vulnerable Components - Using well-maintained crypto libraries</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A07</span>: Identity & Auth Failures - Strong password policy, secure sessions</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A08</span>: Software & Data Integrity Failures - Input validation and sanitization</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A09</span>: Security Logging Failures - Error logging and monitoring</span>
                    </div>
                    <div class="security-item">
                        <span class="check-icon">✓</span>
                        <span class="security-text"><span class="owasp-code">A10</span>: Server-Side Request Forgery - Input validation on all parameters</span>
                    </div>
                </div>
            </div>

            <div class="logout-section">
                <a href="/logout" class="logout-btn">🚪 Logout Securely</a>
            </div>
        </div>
    </div>
</body>
</html>
```

Додаток Б

СКРИПТ, ЩО ЗДІЙСНЮЄ СКАНУВАННЯ НА УРАЗЛИВОСТІ

scan.sh

```
0  #!/bin/bash
1
2  webapp="http://$(ip -f inet -o addr show eth0 | awk '{print $4}' | cut
   -d '/' -f 1):8080"
3  unix_time=$(date +%s)
4  filename="scan-report-$unix_time.html"
5
6  docker run --rm \
7    -v $(pwd):/zap/wrk/:rw \
8    --network="host" \
9    zaproxy/zap-stable \
10   zap-baseline.py \
11   -t $webapp \
12   -r $filename
13
14 echo "Scan report on $webapp is saved to $filename."
15
```