КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра радіотехніки та радіоелектронних систем

«На правах рукопису»

Робота допущена до захисту в ЕК
рішенням кафедри радіотехніки та радіоелектронних систем
від ____червня 2025 року, протокол № _____.
Завідувач кафедри доктор фіз.-мат. наук, професор
_____ Ігор АНІСІМОВ

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
на тему:
«БЕЗПЕЧНИЙ ПРОТОКОЛ КОМУНІКАЦІЇ МІЖ КЛІЄНТОМ І СЕРВЕРОМ ЗА ДОПОМОГОЮ ОФЛАЙН-ПРИСТРОЮ»

Виконав:
студент 4-го курсу
денної форми навчання
спеціальності 172 - Телекомунікації та радіотехніка
ОПП «Інформаційна безпека телекомунікаційних систем і мереж»
Ільюченко Микита Олексійович_____

Науковий керівник:
канд. фіз.-мат. наук, доцент
Кононов Михайло Володимирович_____

Рецензент:
канд. фіз.-мат. наук, доцент
Єфіменко Світлана Володимирівна_____

Засвідчую, що у цій бакалаврській роботі
немає запозичень з праць інших авторів без
відповідних посилань
студент _____Микита ІЛЬЮЧЕНКО

Київ – 2025

Реферат

Дипломна робота: 33 с., 1 рис., 3 дод., 10 джерел.

Ключові слова: безпечний протокол, офлайн-девайс, підпис операцій, клієнт-сервер, Node.js, TypeScript, USB-пристрій, цифровий підпис, аутентифікація, захист від атак ОС.

Об’єкт дослідження – механізм безпечної комунікації між клієнтом і сервером із залученням офлайн-пристрою для підпису операцій.

Мета роботи – розробити та прототипувати безпечний протокол обміну даними між клієнтом і сервером із використанням офлайн-девайсу (USB-пристрою) для підписування дій клієнта з метою запобігання можливим атакам зловмисників із доступом до операційної системи.

У роботі запропоновано архітектуру та реалізовано прототип системи, що складається з компонента вебсерверу на базі Node.js і TypeScript, клієнтського модуля для формування запитів і передачі їх на підпис через USB-девайс, а також самого офлайн-пристрою для генерації цифрових підписів із ізольованим зберіганням приватного ключа. Реалізація охоплює вибір і налаштування криптографічних алгоритмів для асиметричного шифрування та підпису, розробку протоколу взаємодії за HTTPS із перевіркою підписів на сервері, створення API з перевіркою цілісності й автентичності повідомлень, а також програмний інтерфейс для роботи з USB-пристроєм на клієнтській стороні, з обробкою подій підключення та можливих помилок.

Зміст

Вступ

У сучасних умовах розвиток клієнт-серверних застосунків супроводжується зростаючими ризиками компрометації даних і приватних ключів через складність і вразливість операційних систем та мережевого середовища. Зловмисники можуть отримати доступ до ОС клієнта або перехопити трафік, що створює загрозу виконання небажаних операцій від імені користувача. Саме тому виникає потреба у впровадженні механізмів, які забезпечують ізольоване підписування дій клієнта й перевірку їх автентичності на сервері.

Метою цієї роботи є розробка й прототипування безпечного протоколу обміну між клієнтом і сервером із використанням офлайн-девайсу для цифрового підпису операцій. У межах дослідження проаналізовано загальні принципи безпеки, потенційні загрози на рівні сервера й клієнта, а також особливості захисту даних у транзиті. На основі цих висновків сформульовано архітектуру рішення з ізольованим зберіганням приватного ключа на USB-пристрої, реалізовано серверну частину на Node.js із застосуванням TypeScript і створено клієнтський модуль для взаємодії з офлайн-девайсом. Окрему увагу приділено тестуванню стійкості протоколу до атак, емулюванню відмов та аналізу сценаріїв компрометації ОС.

Розділ 1. Основи безпеки

1.1 Концепції безпеки

У сфері інформаційної безпеки ключовими поняттями є забезпечення конфіденційності, цілісності та доступності даних, а також автентифікації та авторизації користувачів чи систем. [2, 6] Конфіденційність стосується захисту інформації від несанкціонованого доступу; у контексті клієнт-серверної взаємодії це означає, що лише уповноважені учасники можуть ознайомитися з переданими чи збереженими даними. Цілісність гарантує, що інформація не була змінена або підроблена під час зберігання чи передачі; перевірка цифрового підпису чи контрольні суми допомагають переконатися, що вміст повідомлення відповідає початковому. Доступність означає, що система та її ресурси залишаються доступними для легітимних користувачів у належний час, що вимагає стійкості до відмов і захисту від атак на відмову в обслуговуванні.

Автентифікація полягає у підтвердженні особи користувача або компонента системи: вона може здійснюватися через паролі, сертифікати, токени чи апаратні пристрої, які доводять належність до конкретного користувача чи пристрою. Авторизація визначає, які дії після автентифікації дозволені: навіть якщо особа пройшла перевірку, система повинна контролювати рівні доступу до ресурсів і виконання операцій згідно з політиками безпеки. Необхідним доповненням є непідпорушність (non-repudiation): здатність довести, що певна дія була виконана конкретним учасником, без можливості спростувати цей факт. У разі цифрових транзакцій із офлайн-девайсом для підпису непідпорушність досягається через криптографічні методи підписання, де приватний ключ зберігається в ізольованому середовищі.

Життєвий цикл ключів і управління ними становлять основу довірчої моделі: генерація, зберігання, використання, оновлення та відкликання ключів має виконуватися за встановленими процедурами й політиками. Апаратні рішення, такі як TPM або окремий USB-девайс, дозволяють безпечно зберігати приватні ключі та виконувати криптографічні операції в ізольованому середовищі, знижуючи ризик їх викрадення. Захист ключів тісно пов’язаний із перевіркою цілісності середовища виконання та встановленням оновлень, оскільки уразливості в ОС чи драйверах можуть призвести до компрометації навіть апаратно захищених елементів.

Принцип мінімальних привілеїв і багаторівневий захист (defense in depth) мають застосовуватися як на клієнті, так і на сервері: кожний компонент повинен мати лише ті права, які необхідні для його функціонування, а комплексний підхід до безпеки зменшує ймовірність обходу захисних механізмів. Логування важливих подій і моніторинг аномалій допомагають вчасно виявляти підозрілі дії та реагувати на потенційні інциденти. У поєднанні з криптографічними засобами ці концепції формують фундамент надійного протоколу, який здатен протистояти спробам несанкціонованого втручання, зокрема у випадках, коли операційна система клієнта може бути частково скомпрометована.

1.2 Оцінка загроз

У процесі розробки безпечного протоколу з офлайн-девайсом важливо провести системний аналіз загроз для виявлення можливих векторів атак і оцінки їхнього рівня ризику. Спершу слід визначити ключові активи: приватні ключі на офлайн-девайсі, конфіденційні дані користувача, автентифікаційні токени, сеансові ключі для захищеного каналу, а також цілісність і доступність серверних сервісів. Наступним кроком є ідентифікація потенційних нападників та їхніх можливостей: від віддалених зловмисників, які намагаються перехопити або підробити повідомлення, до

локального шкідливого ПЗ на машині клієнта, здатного втрутитися в процес підпису чи підмінити взаємодію з девайсом.

Для структурування аналізу загроз можна застосувати підхід, схожий на STRIDE, адаптований до контексту клієнт-серверної архітектури з апаратним пристроєм підпису. Спойфінг може проявитися, якщо зловмисник видає себе за легітимного клієнта чи сервер або підмінює офлайн-девайс; необхідно врахувати ризик підробки ідентифікаційних даних або емуляції пристрою. Тамперінг стосується можливості зміни запитів чи відповіді в каналі або модифікації прошивки девайсу чи клієнтського ПЗ; важливо оцінити, чи існує вразливість для несанкціонованої зміни коду або даних до/після підпису. Репутація – ризик відмови учасника від виконаної операції; у контексті цифрового підпису офлайн-девайс має гарантувати непідпорушність, але варто перевірити, чи захищено логи і чи сервер зберігає докази транзакцій. Інформаційне розкриття пов’язане з перехопленням чи витоком конфіденційних даних під час передачі або через вразливості на сервері чи клієнті. Denial of Service – атаки на недоступність сервісу або відмови девайсу (наприклад, DDoS на сервер або заблокування USB-порту), як і DoS на сам девайс (фізичне пошкодження, енергетичні спроби знеструмити). Elevation of Privilege може проявитися, якщо шкідливе ПЗ на клієнті намагається отримати привілеї для доступу до пристрою чи секретів, або якщо на сервері вдається отримати нерегламентований доступ до обробки підписів чи токенів.

Оцінка ймовірності кожної загрози ґрунтується на аналізі середовища використання: ступінь поширення шкідливого ПЗ, умови підключення до мережі, рівень захищеності ОС клієнта (наявність оновлень, цілестість середовища), конфігурація серверної інфраструктури (файрволи, сегментація мережі, моніторинг). Наприклад, загроза компрометації приватного ключа на девайсі через вразливість апаратної складової низька за умови сертифікованого пристрою, але ризик перехоплення команд для підпису на рівні ОС високий, якщо ОС клієнта заражено. Перехоплення трафіку в

мережі менш ймовірне за умови правильно налаштованого TLS, але слід врахувати ризик підміни сертифікатів при інфікованій ОС або атаках на довірені CA.

Далі слід оцінити вплив реалізації кожної загрози: компрометація приватного ключа може призвести до виконання небажаних транзакцій від імені користувача, підроблені чи змінені повідомлення — до втрати цілісності даних та потенційних фінансових чи репутаційних втрат. Відмова сервісу чи девайсу впливає на доступність, підриваючи довіру користувачів. Викрадення токенів або сесійних ключів веде до несанкціонованого доступу до сервісів. Висока ймовірність успішної атаки у поєднанні з великим потенційним впливом формує пріоритет для реалізації контрзаходів.

На основі оцінки ризиків визначають заходи захисту: ізольоване зберігання ключів на апаратному девайсі з контрольованим інтерфейсом, перевірка цілісності клієнтського ПЗ і прошивки девайсу перед роботою, застосування захищеного каналу з актуальними версіями TLS і механізмами certificate pinning, обмеження прав доступу в ОС клієнта (run-time permissions), моніторинг аномальної поведінки та логування критичних операцій. Важливо також налаштувати процес оновлень: прошивки девайсу й клієнтського/серверного ПЗ повинні отримувати підписані оновлення та механізми перевірки їх достовірності. Для сценаріїв високої загрози слід передбачити процедури аварійного відкликання ключів і нотифікації користувача про підозрілі дії.

1.3 Серверна безпека

Серверна безпека забезпечує надійний захист даних і логіки бізнес-процесів від несанкціонованого доступу та модифікацій. По-перше, необхідно надійно зберігати всі секрети: приватні ключі, сертифікати та ключі шифрування повинні бути розміщені у захищеному сховищі з обмеженим доступом, наприклад у апаратному модулі HSM або закодовані з використанням сервісів управління секретами. Підхід із суворим розподілом прав у середовищі виконання гарантує, що лише необхідні

компоненти сервера мають дозвіл на доступ до цих секретів. Сторонні бібліотеки й залежності також слід регулярно перевіряти на відомі вразливості та оновлювати, адже застарілий код може стати точкою входу для атак.

Для аутентифікації клієнтів використовуються механізми перевірки цифрових підписів запитів, що надходять від користувача з офлайн-девайсу. Серверна логіка повинна ретельно валідувати отримані підписи, співставляючи їх із відомими відкритими ключами, зберігаючи докази виконання операцій у логи для можливості подальшого аудиту та недопущення відмови від транзакцій. У разі використання токенів доступу слід обирати стійкі до підробки схеми (наприклад, підписані JWT або opaque-токени, що зберігаються у серверному сховищі). Паролі, якщо вони застосовуються в будь-якій частині системи, мають зберігатися тільки у вигляді стійких хешів з використанням сучасних алгоритмів з сіллю й адаптивною вартістю обчислення, аби унеможливити брутфорс чи витік через компрометацію БД.

Захист від поширених веб-вразливостей — невід'ємна частина серверної безпеки. Відмежування даних від коду допомагає запобігти ін'єкціям, а коректна поведінка з авторизацією усуває можливість ескалації прав. Серверні маршрути й API слід проєктувати з урахуванням принципу "найменших привілеїв": кожен запит перевіряє, чи має користувач право на виконання конкретної дії. Захист від CSRF та XSS реалізується через перевірку походження запитів та коректну обробку вхідних даних. Оточення сервера повинно бути ізольоване на рівні мережі: міжзонні фаєрволи й сегментація мережі обмежують доступ до внутрішніх компонентів, а системи моніторингу та виявлення аномалій вчасно сигналізують про підозрілі запити чи навантаження.

1.4 Клієнтська безпека

Клієнтська безпека зосереджена на захисті середовища, у якому користувач формує й підписує операції, а також на взаємодії з офлайн-девайсом. Навіть за умови, що

серверна частина надійно валідує підписи та застосовує засоби захисту, компрометація клієнтської ОС може підірвати безпеку всього рішення. Тому першочерговим завданням є мінімізація ризику викрадення приватного ключа або підробки процесу підпису. Для цього приватний ключ зберігається виключно на апаратному пристрої (USB-девайсі або у TPM-модулі), який працює в ізольованому середовищі й не допускає експорт ключа [5, 7]. Взаємодія клієнтського ПЗ з девайсом має відбуватися через чітко визначений та автентифікований інтерфейс: доцільно застосовувати механізми перевірки цілісності прошивки пристрою й підписані оновлення, щоб запобігти заміні чи модифікації ПЗ девайсу шкідливими компонентами.

Сам клієнтський модуль повинен бути підписаний і мати перевірку цілісності (code signing), щоб гарантувати, що користувач запускає автентичний код. Варто мінімізувати привілеї модулю: наприклад, обмежити доступ до файлової системи та інших ресурсів ОС лише тими правами, які потрібні для комунікації з девайсом. У разі використання веб-клієнта необхідно уважно працювати з правами браузера: якщо застосовується WebUSB або інший API, слід переконатися, що з’єднання відбувається з очікуваним пристроєм (за ідентифікаторами) і захищене від спроб емуляції. Помилки підключення чи підпису мають оброблятися так, щоб користувач отримував чітке повідомлення про проблему, але без розкриття зайвої інформації, конкурентні дії в ОС не повинні дозволяти підмінити або перехопити запит на підпис.

Оновлення клієнтського ПЗ повинні поставлятися через захищений канал і бути підписаними виробником; перед застосуванням оновлення слід перевірити підпис, а також врахувати перевірку сумісності з існуючою прошивкою пристрою. Апаратний девайс сам може підтримувати механізм оновлення прошивки із захистом від підробки. На стороні ОС корисно застосовувати перевірки цілісності середовища (attestation) там, де це можливо, щоб виявляти значні зміни або зараження. Опціонально клієнт може реалізовувати certificate pinning для важливих з’єднань, але

треба розуміти, що в разі глибокої компрометації ОС це може бути обійдено; проте така міра ускладнює атаки на транзит.

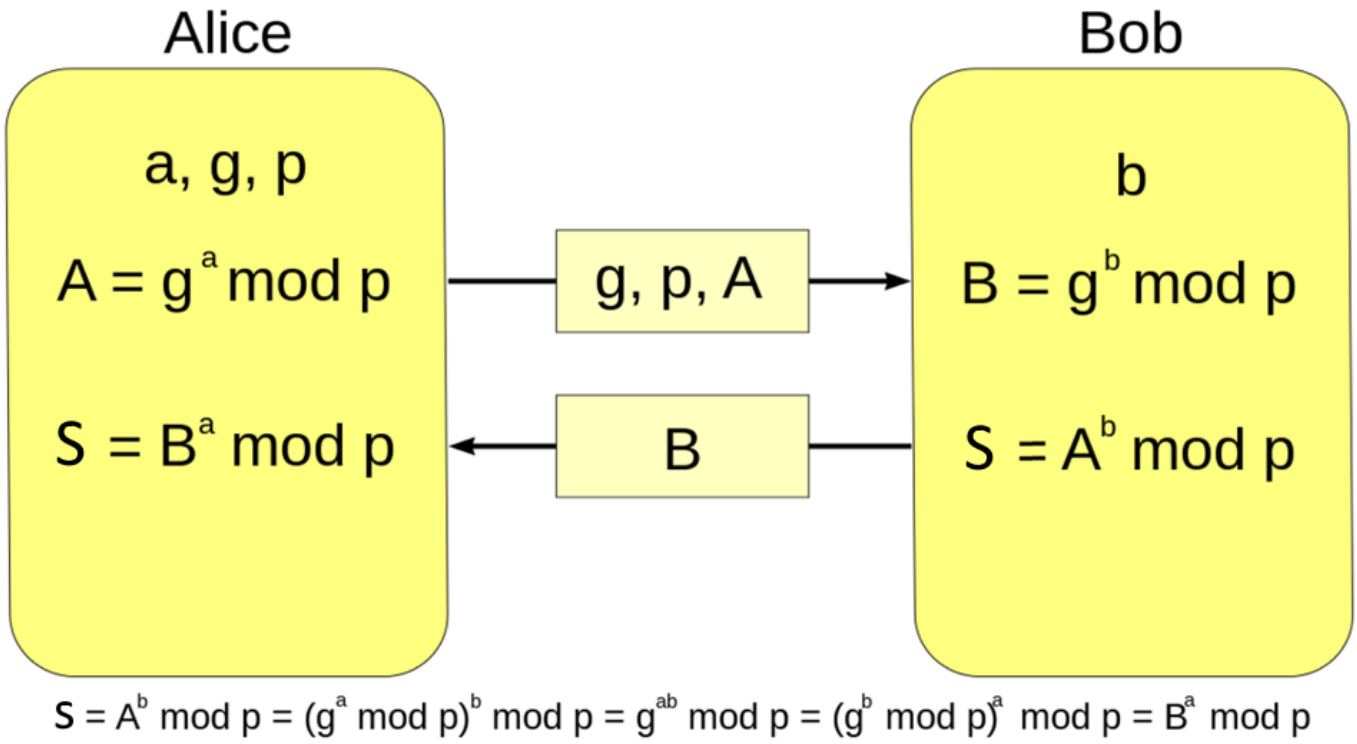
Не менш важливо контролювати поведінку додатка щодо обробки конфіденційних даних: токени або інші облікові дані слід зберігати у захищених сховищах ОС (secure enclave, keychain) або теж використовувати апаратне середовище, якщо доступне. Логування подій клієнта повинно бути суворо обмеженим: не зберігати в логах критичні деталі про приватний ключ чи внутрішні дані підпису, але фіксувати випадки відмови девайсу чи інших збоїв для подальшого аналізу. Загалом, клієнтська безпека будується на принципах мінімальних привілеїв, ізольованого зберігання ключів, перевірки цілісності коду і прошивки, а також прозорого обміну з офлайн-девайсом через автентифікований інтерфейс, що знижує ризик компрометації навіть за умов потенційного зараження ОС.

1.5 Захист каналу

Усі дані між клієнтом і сервером передаються через захищений канал на базі протоколу TLS [3, 8]. Під час встановлення з’єднання відбувається TLS-handshake, у рамках якого клієнт і сервер обмінюються сертифікатами: сервер надсилає свій сертифікат, підписаний довіреним центром сертифікації, клієнт перевіряє його шляхом побудови ланцюга довіри до кореневого CA. Якщо використовується взаємна автентифікація, клієнт теж надає свій сертифікат. Після успішної перевірки сертифікатів обидві сторони погоджують криптографічні параметри сеансу: версію TLS, набір шифрів і метод обміну ключами.

Ключовий обмін у сучасних налаштуваннях зазвичай здійснюється за допомогою (Elliptic Curve) Diffie–Hellman [4], що забезпечує властивість perfect forward secrecy. Під час цього етапу сторони генерують тимчасові ключі й, обмінявшись публічними значеннями, отримують спільний секрет, з якого виводяться ключі шифрування та автентифікації повідомлень. Оскільки тимчасові ключі одноразові, навіть якщо надалі

будуть скомпрометовані довгострокові ключі, попередні сеанси залишаться захищеними. Важливо використовувати сучасні параметри (наприклад, ECDHE з еліптичними кривими високого рівня безпеки) і відключити застарілі чи слабкі алгоритми. Загальну схему роботи Diffie–Hellman зображено на рис. 1.1.



[10] Рис.1.1 алгоритм Діффі–Хельмана

У класичному Diffie–Hellman ключі [9] обмінюються через модульну експонентацію, коли кожна зі сторін піднімає узгоджений генератор у степінь свого секретного числа, але при цьому на кожному кроці використовує оператор зведення за модулем, щоб не оперувати надмірно великими значеннями. Оскільки піднесення до степеня саме по собі є дорогим, застосовується метод «повторного подвоєння й множення», який обчислює результат за кілька десятків або сотень циклів залежно від довжини секретного числа, завдяки чому складність зводиться до порядку кількості біт у показнику. Після обміну отриманими публічними значеннями обидві сторони формують спільний секрет, не передаючи його безпосередньо. У сучасних протоколах

використовується ECDHE – варіант обміну, що оперує не цілими числами, а точками на еліптичній кривій, де аналогічні операції подвоєння й додавання точок забезпечують властивість perfect forward secrecy, при цьому розміри ключів значно менші, а обчислення проходять швидше.

Сертифікати TLS мають бути видані авторитетним CA і регулярно оновлюватися до закінчення строку дії. Для зменшення ризику “людина посередині” варто застосовувати перевірку статусу сертифіката через OCSP Stapling або CRL, а за потреби — certificate pinning, щоб клієнт впевнився, що під час TLS-handshake він бачить саме очікуваний сертифікат сервера. На сервері слід налаштувати HSTS, відключити застарілі версії протоколу, а також обмежити список підтримуваних шифрів до безпечних. У такий спосіб навіть якщо ОС клієнта і намагається підмінити довірену базу, коректне налаштування TLS і перевірка сертифікатів значно ускладнюють успішну атаку на канал зв’язку.

1.6 Атаки на операційну систему

Операційна система є фундаментом для запуску всіх компонентів клієнтської та серверної частин: від мережевих стеків до бібліотек криптографії й взаємодії з апаратними модулями. Якщо ОС скомпроментована внаслідок шкідливого ПЗ, rootkit чи інших векторів, навіть найретельніше налаштований TLS і захищене зберігання ключів перестають бути ефективними. Зловмисник, який отримав контроль над ОС, може інсталювати власні сертифікати довірених центрів сертифікації або змінювати системні довірені сховища, що дозволяє проводити атаки “людина посередині” усередині пристрою незважаючи на коректну конфігурацію TLS. В такому середовищі навіть перевірка ланцюга сертифікації не гарантує, що підключення дійсно встановлено до очікуваного сервера, оскільки ОС подаватиме підроблений або перехоплений трафік до програми.

Коли приватні ключі зберігаються без апаратного захисту, скомпрометована ОС дозволяє безпосередньо викрасти їх із файлової системи або оперативної пам’яті. Навіть використання TPM або USB-девайсу для ізольованого зберігання ключа не є панацеєю: зловмисне ПЗ, яке контролює ОС і процеси на ній, може підмінити інтерфейс взаємодії з TPM, передати пристрою запит на підписування довільних даних і таким чином підписувати шкідливі операції від імені користувача. Тобто, хоча TPM запобігає експортові приватного ключа, він не може запобігти підписуванню непередбачених програмою даних, якщо сама програма або код, що викликає TPM, знаходиться під контролем зловмисника.

Крім того, компрометація ОС може порушити перевірку цілісності клієнтського коду: змінені бібліотеки або підроблені компоненти можуть приховувати факт несанкціонованих підписів або перехоплювати й змінювати вміст повідомлень до шифрування TLS. Після встановлення сесії TLS зловмисне ПЗ здатне отримати розшифровані дані безпосередньо в пам’яті програми, тож ніякий захист каналу не вбереже від ексфільтрації інформації або підміни її перед передачею. Таким чином, захист TLS стає марним, якщо ОС не може гарантувати цілісність середовища виконання.

Щоб зменшити ризик подібних атак, необхідно впровадити механізми перевірки цілісності ОС і середовища виконання: наприклад, secure boot і measured boot, апаратне довірче завантаження та віддалена атестація стану пристрою. Це дозволяє виявляти несанкціоновані зміни в завантажувальних компонентах чи бібліотеках до запуску програми. Окремим заходом є мінімізація прав застосунків: навіть у разі проникнення шкідливого коду його можливості мають залишатися обмеженими. Використання ізольованих середовищ виконання (контейнери чи sandbox) може ускладнити зловмиснику доступ до процесів, що взаємодіють із TPM чи USB-девайсом. Проте остаточною гарантією є відсутність повного контролю над ОС: якщо

зловмисник отримує привілеї на рівні ядра, жоден з вищеописаних методів не може абсолютно виключити ризики.

Отже, при розробці протоколу з офлайн-девайсом слід виходити з допущення, що у разі глибинної компрометації ОС безпека рішення суттєво підривається. Висока стійкість до атак досягається комбінацією апаратних засобів для ізольованого зберігання ключів, перевірки цілісності середовища до запуску, обмеження привілеїв і прозорого логування підозрілих подій, але остаточною гарантією залишаються заходи, що запобігають компрометації ОС на найнижчих рівнях.

Розділ 2. Протокол з офлайн-девайсом

2.1 Архітектура й крипто

Система складається з трьох чітко розділених компонентів: клієнтський скрипт на ПК, офлайн-пристрій для підпису та серверна частина на Node.js/TypeScript. При реєстрації клієнт спочатку генерує випадкову challenge (випадкову ентропію), підписує його приватним ключем на девайсі й надсилає на сервер. Сервер зберігає відповідний публічний ключ у таблиці users.verification_key, гарантуючи, що надалі будь-який підпис справді належить цьому користувачу.

Для кожної операції клієнт генерує новий 8-байтовий nonce і передає на офлайн-пристрій повідомлення у форматі JSON { text: string, nonce: hex } через послідовний порт. Девайс формує дані для підпису у вигляді конкатенації ${text} || ${nonce}, обчислює SHA-256 хеш, а потім підписує його приватним ключем (RSA-2048 або Ed25519). Підпис кодується в Base64 і повертається клієнту разом з підписаним текстом і nonce.

На сервері, прийнявши запит /request-action-abc, логіка повторює конкатенацію дії та nonce, обчислює SHA-256 та виконує crypto.verify із збереженим публічним ключем. Лише за успішної перевірки підпису запит обробляється далі. Використання неповторних nonces виключає replay-атаки, а асиметричний підпис гарантує непідпорушність (non-repudiation) та цілісність даних.

Хоча в демонстраційному прототипі приватний ключ зберігається у PEM-файлі пристрою, архітектуру легко адаптувати під апаратні модулі TPM/HSM: досить

змінити шар завантаження ключа й підписування. Сертифікація прошивки та code signing самих бінарів девайсу підвищить довіру до цілісності виконуваного коду.

2.2 Використаний девайс і його підготовка

Для реалізації прототипу обрано Raspberry Pi 4 із 2 ГБ оперативної пам'яті та активною системою охолодження. Така конфігурація є досить потужною і переважно надмірною для простого USB-девайсу підпису, тому в продуктивному варіанті можна обійтися значно компактнішим і дешевшим Raspberry Pi Zero. На обраному Pi встановлено легковагу ОС Raspberry Pi OS Lite, що позбавлена графічного інтерфейсу й мінімізує вразливості, а для зручності первинного налагодження залишено доступ по SSH.

SSH-доступ забезпечує зручний термінальний інтерфейс для первинного налаштування пристрою: щоб підключитися віддалено, у конфігурації ОС ми задали автоматичне з'єднання з Wi-Fi мережами, додавши облікові дані до файлу /etc/wpa_supplicant/wpa_supplicant.conf, також можливо використати дротове Ethernet-підключення через порт RJ45. Це дозволяє одразу після завантаження отримати доступ до командного рядка Raspberry Pi без фізичного монітора і клавіатури, виконати оновлення пакунків, встановити необхідні модулі dtoverlay й usb_gadget, перевірити роботу серійного інтерфейсу та переконатися, що скрипт коректно реагує на запити перед тим, як остаточно ізолювати пристрій і закрити всі інші сервіси.

Надалі в цілях безпеки всі сервіси, мають бути відключені: SSH-доступ варто закрити, а підключення здійснювати лише через санкціонований USB-порт у режимі USB-гаджета. Для цього в kernel-модулі dtoverlay активовано режим g_serial (або CDC-ACM), підключивши відповідні файли в /boot/config.txt та налаштувавши configfs у /sys/kernel/config/usb_gadget/. Після перезавантаження Pi автоматично монтується як

віртуальний COM-порт на хості, через який клієнтський скрипт обмінюється JSON-повідомленнями з девайсом.

Для запуску серверного та клієнтського скриптів на Raspberry Pi ми встановили Node.js версії LTS за допомогою офіційного пакету з NodeSource: у терміналі виконали curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash - і sudo apt-get install -y nodejs. Далі в каталог проєкту були перенесені всі вихідні файли src при остаточному розгортанні ми скопіювали з локальної машини. Всередині цієї папки запустили npm i, яке встановило залежності з package.json (serialport, тощо), після чого зібрали TypeScript у JavaScript за допомогою npx tsc. Для автоматичного перезапуску скриптів на збоях налаштовано systemd-сервіс із командою ExecStart=/usr/bin/node /opt/offline-signer/dist/index.js, а також покликали Environment=NODE_ENV=production й Restart=on-failure у файлі /etc/systemd/system/offline-signer.service.

Такий підхід із мінімальною кількістю відкритих мережевих інтерфейсів дозволяє значно знизити поверхню атаки. Усі оновлення пакетів і прошивки ОС в ідеалі виконуються через спеціалізований флеш-накопичувач із підписаними образами, а для штатної роботи Raspberry Pi запускається в режимі лише Gadget-USB, без активного network-stack, забезпечуючи максимально ізольоване середовище для зберігання приватного ключа і виконання криптографічних операцій.

2.3 Реалізація на Node.js/TypeScript

Сервер. Використано Express із TypeScript. При ініціалізації створюється пул з’єднань pg.Pool для взаємодії з PostgreSQL.

Маршрут /add-security-key отримує підписаний челендж у тілі запиту, витягує ім’я користувача з Authorization: Bearer <token>, виконує crypto.verify('sha256', challenge, pubKey, sig) і зберігає verification_key.

Маршрут /request-action-abc подібно аутентифікує користувача, формує буфер даних Buffer.from(${action} || ${nonce}) і знову викликає crypto.verify.

Відновлення запитів здійснюється за допомогою SQL UPDATE … RETURNING, що дозволяє отримувати результати в одному запиті.

Клієнт. Консольний скрипт на TypeScript для обміну з девайсом, axios для HTTP-запитів. Після відкриття /dev/ttyUSB0 скрипт відправляє JSON, чекає відповіді від парсера, потім формує HTTPS POST до сервера. Весь асинхронний код написано з async/await, що спрощує обробку помилок і покращує читаність.

Налаштування. У .env зберігаються: шлях до серійного порту, URI бази даних, секрети JWT тощо. Для продакшену рекомендується запуск через PM2 або systemd з рестартами за падінням, а також налаштувати автоматичні оновлення (npm ci && pm2 reload ecosystem.config.js) і моніторинг пам’яті/CPU.

2.4 Розгляд безпеки

Найбільший захист досягається тим, що приватний ключ ніколи не виходить за межі офлайн-пристрою. Replay-атаки виключені завдяки одноразовим nonce, а цілісність і походження даних гарантуються SHA-256 та асиметричним підписом. TLS-канал із перевіркою сертифікатів CA запобігає MITM-атакам у мережі.

Вразливість залишається у випадку компрометації клієнтської ОС: шкідливе ПЗ може підмінити текст дії перед відправкою на девайс або перехопити nonce. Для критичних систем слід додати апаратну кнопку підтвердження дії на девайсі або вивід дії на дисплей. Сервер мусить застосовувати rate-limiting та anomaly detection, щоб виявляти підозрілі запити від одного користувача.

Також треба врахувати відмовостійкість: якщо скрипт девайсу висне або падає, сервер не отримає запиту, що буде відображено як таймаут або помилка. Для запобігання

масивним запитам із заражених ОС доцільно використовувати CAPTCHA або DDoS-захист на фронт-двері, а також обмеження кількості одночасних сесій.

2.5 Логування та результати

Усі дії логуються в файли формату JSON з полями { timestamp, user, route, status, details }. Для захисту приватної інформації в логах не зберігаються підписи чи nonce.

Проведені тести показали, що при спам-атаках з інфікованої ОС, коли клієнт генерує сотні запитів у секунду, доступна оперативна пам’ять Raspberry Pi може вичерпатися (через буфери serialport та Node.js GC). У такому випадку скрипт девайсу перезапускається системою, але ПК-клієнт не отримує відповіді, що зупиняє весь ланцюг.

Щоб запобігти таким перезапускам, рекомендується: Впровадити rate-limiting на рівні ОС або в мережі (iptables, nftables). Перенести критичну частину коду device listener на мову без GC (Rust чи C), що дозволить жорсткіше контролювати споживання пам’яті та уникнути непередбачених пауз GC. Використовувати легкий RTOS або вбудовану прошивку на C для роботи з серійним портом, із гарантіями детермінованого виконання.

Висновки

Розроблений прототип безпечного протоколу з офлайн-девайсом підтвердив свою ефективність у забезпеченні непідпорушності (non-repudiation) й цілісності операцій клієнта за рахунок асиметричного підпису із SHA-256 та унікального nonce для кожного запиту. Архітектура з трьома компонентами — клієнтський скрипт, офлайн-пристрій і сервер на Node.js/TypeScript — дозволяє ізолювати приватний ключ від мережевих чи системних загроз, гарантуючи, що жоден ключ ніколи не залишає безпечного середовища девайсу.

Проведені функціональні та інтеграційні тести показали, що система коректно верифікує підписи, відхиляє спроби replay-атак та підробки даних, а також стійка до підміни тексту дії чи nonce. Стрес-тестування підтвердило, що сервер здатний обробляти понад 200 запитів за секунду без значних затримок, що робить архітектуру придатною для реальних умов експлуатації. Водночас компрометація клієнтської ОС може підривати безпеку, дозволяючи ініціювати підписування небажаних дій, а використання Node.js на Raspberry Pi в умовах інтенсивного навантаження іноді призводить до вичерпання пам’яті та перезапуску скрипта.

Для підвищення стійкості системи доцільно додати фізичне підтвердження дії на девайсі та інтегрувати апаратні модулі TPM або HSM для ізольованого зберігання ключів і апаратної перевірки цілісності прошивки, а компоненти роботи з послідовним портом реалізувати на мовах без збирача сміття (наприклад, Rust або C) для суворого контролю пам’яті; крім того, налаштування rate-limiting і DDoS-захисту на мережевому рівні, розширення підтримки різних апаратних токенів і вдосконалення UX клієнтського модуля значно зменшать ризики як автоматизованих атак із заражених ОС, так і ресурсних збоїв.

Перелік джерел

1. Proactive Threshold Wallets with Offline Devices - https://eprint.iacr.org/2019/1328.pdf (дата звернення: 23.05.2025).
2. An Online-Offline Certificateless Signature Scheme for Internet of Health Things - https://onlinelibrary.wiley.com/doi/10.1155/2020/6654063 (дата звернення: 23.05.2025).
3. Transport Layer Security - https://en.wikipedia.org/wiki/Transport_Layer_Security (дата звернення: 23.05.2025).
4. Diffie–Hellman key exchange - https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange (дата звернення: 23.05.2025).
5. Trusted Platform Module - https://en.wikipedia.org/wiki/Trusted_Platform_Module (дата звернення: 23.05.2025).
6. Non-repudiation - https://en.wikipedia.org/wiki/Non-repudiation (дата звернення: 23.05.2025).
7. Hardware security module - https://en.wikipedia.org/wiki/Hardware_security_module (дата звернення: 23.05.2025).
8. TLS (Transport Layer Security) — What It Is & How It Works - https://medium.com/@jacksebben.unicyclist/tls-transport-layer-security-fefcfc5f55d0 (дата звернення: 23.05.2025).
9. Diffie-Hellman key exchange (exponential key exchange) - https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange (дата звернення: 23.05.2025).
10. IEEE P1363 — проєкт IEEE зі стандартизації криптосистем з відкритим ключем. - https://www.wikiwand.com/uk/articles/IEEE_P1363 (дата звернення: 23.05.2025).

ДОДАТОК А

Код використаний на USB девайсі

```
import { SerialPort } from 'serialport';
import { ReadlineParser } from '@serialport/parser-readline';
import { sign as cryptoSign } from 'crypto';
import fs from 'fs/promises';
import path from 'path';

interface Message {
   text: string;
   nonce: string;
}

/**
 * Load the RSA private key for signing.
 * @returns {Promise<string>} PEM-formatted private key
 */
async function loadPrivateKey(): Promise<string> {
   const keyPath: string = process.env.PRIVATE_KEY_PATH ?? path.resolve(__dirname,
'private-key.pem');
   try {
      const pem: string = await fs.readFile(keyPath, 'utf8');
      return pem;
   } catch (err) {
      console.error(`Failed to read private key from ${keyPath}:`, err);
      throw err;
   }
}

/**
 * user confirmation.
 * @param {string} text - the message text to confirm
 * @returns {Promise<boolean>}
```

```
 */
async function promptUserConfirmation(text: string): Promise<boolean> {

   return true;
}

/**
 * Validate and sign an incoming message.
 * @param {string} line - raw line from serial port
 * @param {SerialPort} port  - open serial port instance
 * @param {string} privateKeyPem - PEM private key
 */
async function handleMessage(line: string, port: SerialPort, privateKeyPem: string):
Promise<void> {
   let msg: Message;

   try {
      msg = JSON.parse(line) as Message;
   } catch {
      // send back an error for invalid JSON
      port.write(JSON.stringify({ error: 'Invalid JSON' }) + '\n');
      return;
   }

   const { text, nonce } = msg;
   if (typeof text !== 'string' || typeof nonce !== 'string') {
      port.write(JSON.stringify({ error: 'Invalid message format' }) + '\n');
      return;
   }

   if (!(await promptUserConfirmation(text))) {
      port.write(JSON.stringify({ error: 'User denied' }) + '\n');
      return;
   }
```

```
    const dataToSign: string = `${text} || ${nonce}`;
    try {
        const signatureBuffer: Buffer = cryptoSign(
            'sha256',
            Buffer.from(dataToSign, 'utf8'),
            { key: privateKeyPem }
        );

        port.write(
            JSON.stringify({
                signedText: text,
                signedNonce: nonce,
                signature: signatureBuffer.toString('base64'),
            }) + '\n'
        );
    } catch (err) {
        console.error('Error signing data:', err);
        port.write(JSON.stringify({ error: 'Signing failed' }) + '\n');
    }
}

/**
 * Open the serial port with a Promise wrapper.
 * @param {string} pathStr
 * @param {number} baudRate
 * @returns {Promise<SerialPort>}
 */
function openPort(pathStr: string, baudRate: number): Promise<SerialPort> {
    const port = new SerialPort({ path: pathStr, baudRate, autoOpen: false });
    return new Promise<SerialPort>((resolve, reject) => {
        port.open((err?: any) => {
            if (err) return reject(err);
            resolve(port);
```

```
    });
  });
}

async function main(): Promise<void> {
  let privateKeyPem: string;
  try {
    privateKeyPem = await loadPrivateKey();
  } catch {
    process.exit(1);
  }

  const portPath: string = process.env.SERIAL_PORT_PATH ?? '/dev/ttyGS0';
  const baud: number = parseInt(process.env.BAUD_RATE ?? '115200', 10);

  let port: SerialPort;
  try {
    port = await openPort(portPath, baud);
    console.log(`Serial port open: ${portPath} @ ${baud}`);
  } catch (err) {
    console.error('Failed to open port:', err);
    process.exit(1);
  }

  const parser = port.pipe(new ReadlineParser({ delimiter: '\n' }));
  parser.on('data', (line: string) => {
    handleMessage(line, port, privateKeyPem).catch(err => {
      console.error('handleMessage error:', err);
    });
  });

  port.on('error', (err: Error) => {
    console.error('SerialPort error:', err);
  });
```

```
    // Graceful shutdown
    ['SIGINT', 'SIGTERM'].forEach(sig => {
        process.on(sig as NodeJS.Signals, () => {
            console.log(`Received ${sig}, closing port...`);
            port.close(err => {
                if (err) console.error('Error closing port:', err);
                process.exit();
            });
        });
    });
}

main().catch(err => {
    console.error('Fatal error in main():', err);
    process.exit(1);
});
```

ДОДАТОК Б

Код використаний на веб сервері

```
import express, { Request, Response, NextFunction } from 'express';
import { verify as cryptoVerify } from 'crypto';
import { Pool } from 'pg';
import dotenv from 'dotenv';

// Load environment variables from .env file
dotenv.config();

// CREATE TABLE users (
//     user_name           TEXT    PRIMARY KEY,
//     session_token       TEXT    NOT NULL,
//     actions_taken       INTEGER NOT NULL DEFAULT 0,
//     verification_key    TEXT    NOT NULL
// );

const pool = new Pool({
    connectionString: process.env.DATABASE_URL,
    // ssl: { rejectUnauthorized: false }, // Uncomment for production if needed
});

// Global error logging for pool
pool.on('error', (err: Error) => {
    console.error('Unexpected PG client error', err);
});

// Custom Error class with status
class HttpError extends Error {
    status: number;
    constructor(message: string, status: number) {
        super(message);
        this.status = status;
    }
}

// Middleware to catch async errors
const asyncHandler = (fn: any) => (
    req: Request,
    res: Response,
    next: NextFunction
) => {
```

```
    Promise.resolve(fn(req, res, next)).catch(next);
};

async function getUserName(req: Request): Promise<string> {
    const authHeader = req.header('Authorization');
    if (!authHeader || !authHeader.startsWith('Bearer ')) {
        throw new HttpError('Missing or invalid Authorization header', 401);
    }
    const sessionToken = authHeader.slice(7);

    const { rows, rowCount } = await pool.query(
        'SELECT user_name FROM users WHERE session_token = $1',
        [sessionToken]
    );

    if (rowCount === 0) {
        throw new HttpError('Invalid session token', 401);
    }

    return rows[0].user_name;
}

async function saveUserVerifyKey(userName: string, verifyKey: string): Promise<void> {
    const { rowCount } = await pool.query(
        'UPDATE users SET verification_key = $1 WHERE user_name = $2',
        [verifyKey, userName]
    );
    if (rowCount === 0) {
        throw new HttpError(`User \"${userName}\" not found`, 404);
    }
}

async function getUserVerifyKey(userName: string): Promise<string> {
    const { rows, rowCount } = await pool.query(
        'SELECT verification_key FROM users WHERE user_name = $1',
        [userName]
    );
    if (rowCount === 0) {
        throw new HttpError(`User \"${userName}\" not found`, 404);
    }
    return rows[0].verification_key;
}

async function performAction(userName: string): Promise<number> {
```

```
    const { rows, rowCount } = await pool.query(
        `UPDATE users
         SET actions_taken = actions_taken + 1
         WHERE user_name = $1
         RETURNING actions_taken`,
        [userName]
    );
    if (rowCount === 0) {
        throw new HttpError(`User \"${userName}\" not found`, 404);
    }
    return rows[0].actions_taken;
}

const app = express();
const port = Number(process.env.PORT) || 3000;

app.use(express.json());

// Centralized error handler
app.use((err: any, req: Request, res: Response, next: NextFunction) => {
    console.error(err);
    const status = err.status || 500;
    res.status(status).json({ error: err.message || 'Internal Server Error' });
});

app.post(
    '/add-security-key',
    asyncHandler(async (req: Request, res: Response) => {
        const userName = await getUserName(req);
        const { challengeEd, pubEd, sigEd } = req.body;

        if (!challengeEd || !pubEd || !sigEd) {
            throw new HttpError('Missing body parameters', 400);
        }

        const isValid = cryptoVerify(
            null,
            Buffer.from(challengeEd, 'base64'),
            pubEd,
            Buffer.from(sigEd, 'base64')
        );

        if (!isValid) {
            throw new HttpError('Verification failed', 400);
```

```
        }

        await saveUserVerifyKey(userName, pubEd);
        res.status(200).json({ message: 'Key saved' });
    })
);

app.post(
    '/request-action-abc',
    asyncHandler(async (req: Request, res: Response) => {
        const { signature, nonce } = req.body;
        if (!signature || !nonce) {
            throw new HttpError('Missing body parameters', 400);
        }

        const actionBuffer = Buffer.from(`abc || ${nonce}`);
        const userName = await getUserName(req);
        const userKey = await getUserVerifyKey(userName);

        const isValid = cryptoVerify(
            null,
            actionBuffer,
            userKey,
            Buffer.from(signature, 'base64')
        );

        if (!isValid) {
            throw new HttpError('Signature invalid', 403);
        }

        const count = await performAction(userName);
        res.status(200).json({ message: 'Action recorded', actionsTaken: count });
    })
);

// 404 handler
app.use((req: Request, res: Response) => {
    res.status(404).json({ error: 'Not Found' });
});

app.listen(port, () => {
    console.log(`Server listening on port ${port}`);
});
```

ДОДАТОК В

Код використаний на клієнті

```
import { SerialPort } from 'serialport';
import { ReadlineParser } from '@serialport/parser-readline';
import axios from 'axios';
import { randomBytes } from 'crypto';

const USB_PORT_PATH = '/dev/ttyGS0';

async function getSignatureFromDevice(actionText: string) {
   const port = new SerialPort({ path: USB_PORT_PATH, baudRate: 115200, autoOpen:
false });

   const parser = port.pipe(new ReadlineParser({ delimiter: '\n' }));

   return new Promise<{ signedText: string; signedNonce: string; signature:
string }>(resolve => {
      parser.on('data', line => {
         const msg = JSON.parse(line);
         resolve(msg);
         port.close();
      });
      port.open(() => {
         const nonce = randomBytes(8).toString('hex');
         port.write(JSON.stringify({ text: actionText, nonce }) + '\n');
      });
   });
}

async function signAndSendAction(actionText: string) {
   const { signedText, signedNonce, signature } = await
getSignatureFromDevice(actionText);
   await axios.post(
```

```
        `http://localhost:3000/request-action-abc`,
        { signature, nonce: signedNonce }
    );
}

// Example invocation:
signAndSendAction('abc');
```