НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра математичних методів захисту інформації

Звіт

з комп’ютерного практикуму

«Дослідження особливостей реалізації існуючих програмних систем, які використовують криптографічні механізми захисту інформації»

з кредитного модуля

«Сучасні алгебраїчні криптосистеми»

Варіант 1В

Виконала студентка

Групи ФІ-52мн

Остаповець Олеся

Київ 2025

Основні задачі програмної реалізації криптосистем

Програмна реалізація криптосистем відрізняється від “теоретичного” опису алгоритмів тим, що безпека залежить не лише від математичної стійкості, а й від того, **як саме** алгоритм реалізовано в коді та в якому середовищі він виконується. На практиці криптографічна програма працює в операційній системі, використовує оперативну пам’ять, взаємодіє з файлами, бібліотеками, драйверами та пристроями, а також виконується в умовах потенційних атак з боку шкідливого ПЗ або користувача з підвищеними правами. Тому основні задачі реалізації криптосистеми можна розглядати як комплекс технічних і організаційних заходів, що забезпечують конфіденційність ключів, коректність криптографічних операцій та контроль середовища виконання.

Ключова інформація (секретні ключі, сеансові ключі, паролі та секрети для їх отримання, проміжні значення) проходить певний життєвий цикл: **генерація → зберігання → завантаження в пам’ять → використання → знищення**. На кожному етапі виникають ризики витоку або підміни. Особливо критичним є етап використання, оскільки під час криптографічних операцій ключ або похідні від нього значення **неминуче з’являються в оперативній пам’яті**. Відповідно, завдання захисту ключів включає не лише правильне зберігання на диску чи в сховищі, а й контроль доступу до даних у RAM та мінімізацію часу їх перебування в пам’яті.

Типові загрози

- Компрометація ключів у RAM: читання пам’яті процесу, створення дампів, витік через помилки в обробці пам’яті.
- Ін’єкція або втручання в процес: підміна коду в пам’яті, ін’єкція бібліотеки, перехоплення викликів криптобібліотеки, витяг ключових даних через debug-інтерфейси.
- Підміна або компрометація залежностей: використання модифікованих криптобібліотек, підміна DLL/so, атаки через ланцюг постачання (supply chain) або неконтрольовані оновлення.
- Помилки реалізації та неправильні налаштування: використання слабких параметрів, некоректних режимів, повторне використання nonce, неправильна обробка помилок (наприклад, “ігнорування” невдалої перевірки підпису/тега).
- Побічні канали та витоки через поведінку програми: залежність часу виконання від секретних даних, витік через повідомлення про помилки, логування чутливої інформації.

Також доцільно було б розглянути моделі зловмисника:

- Зовнішній зловмисник без доступу до ОС – впливає лише через мережу, вхідні дані.
- Локальний користувач без адміністративних прав – може запускати процеси, намагатися читати файли у межах своїх прав.
- Шкідливе ПЗ або зловмисник з правами адміністратора – здатен створювати дампи, втручатися в процеси, змінювати політики системи, підміняти бібліотеки.

Контроль доступу до ключової інформації

Під час роботи криптографічної програми секретні ключі або похідні від них значення неминуче з'являються в оперативній пам'яті: у стеку, у буферах бібліотек, у внутрішніх структурах криптопровайдера, а також у кешах процесу. Навіть якщо ключ зберігається на диску в зашифрованому вигляді або знаходиться в апаратному сховищі, на етапі використання часто виникає потреба у розшифруванні ключа, виконанні криптооперації та подальшому збереженні проміжних результатів у пам'яті. Тому задача захисту ключів включає окремий напрям – обмеження можливості зчитування ключової інформації з RAM.

Найпоширенішими сценаріями витоку ключів з RAM у середовищі кінцевого пристрою є:

- Дамп пам'яті процесу (через системні засоби, debugger, інструменти аналізу, аварійні дампи).
- Збереження в swap/hibernation (частина RAM може потрапляти на диск у файл підкачки або файл гібернації).
- Ін'єкція в процес / читання пам'яті процесу (шкідливе ПЗ або користувач з високими правами може отримувати доступ до адресного простору процесу).
- Неправильне поводження з пам'яттю в програмі (копії ключів у кількох буферах, відсутність очищення, використання рядків/логів тощо).

Ці загрози особливо критичні, якщо зловмисник має локальний доступ або права адміністратора.

Для зменшення ризиків компрометації ключів у RAM застосовують такі підходи:

1. Мінімізація часу життя ключа в пам'яті.
   Ключові дані завантажуються в RAM лише на час виконання операції,

після чого одразу видаляються. Це зменшує вікно атаки, тобто час, коли можна провести атаку.

2. Zeroization (безпечне очищення пам’яті).
   Після використання ключові буфери необхідно очищувати (перезаписувати нулями або випадковими даними), уникаючи ситуацій, коли оптимізації компілятора викидають очищення як непотрібне.
3. Зменшення кількості копій ключа.
   Не дублювати ключ у різних структурах, передавати його як звичайний рядок, не робити зайві серіалізації/десеріалізації. Краще працювати з контейнером ключа.
4. Ізоляція та принцип найменших привілеїв.
   Криптографічні операції бажано виконувати в окремому процесі/сервісі з мінімальними правами; основний застосунок взаємодіє з ним через IPC. Це ускладнює ін’єкцію та зчитування пам’яті.
5. Використання апаратних або зовнішніх сховищ ключів.
   TPM, HSM, токени, смарткарти дозволяють виконувати операції підпису/розшифрування всередині модуля так, що ключ не експортується у RAM прикладного процесу. Це один із найефективніших способів зниження ризику.
6. Обмеження дампів і дебагу.
   Де можливо – забороняти або обмежувати створення дампів для чутливих процесів, мінімізувати crash reporting з чутливими даними, обмежувати доступ до інструментів дебагу.

Особливості Windows як endpoint-середовища полягають у тому, що криптографічні операції часто виконуються прикладними програмами через системні компоненти, а також широко використовується централізоване адміністрування (політики безпеки, контроль прав користувачів). Водночас у Windows поширені сценарії, коли зловмисник отримує локальні права, після чого стають реалістичними атаки через дампи процесів, дебаг та ін’єкції. Тому для Windows доцільно комбінувати такі підходи:

- використання системних механізмів роботи з ключами (наприклад, через криптопровайдери ОС), щоб ключі зберігалися в захищених сховищах і за можливості використовувались через handle, а не як “сирі” байти;
- запуск криптографічних компонентів від окремого облікового запису з мінімальними правами;
- обмеження інструментів, що дозволяють знімати дампи/дебажити процеси, для звичайних користувачів; контроль локальних адміністраторів;

- зменшення ризику витоку в файл підкачки/на диск через політики безпеки, шифрування диску та контроль конфігурації.

Отже, ключова ідея – максимально спиратися на системний криптопровайдер і обмежувати локальні привілеї та інструменти аналізу процесів

Для Linux типовою є модель, де безпека значною мірою будується на чіткому розмежуванні прав користувачів і процесів, а також на можливості докрутити ізоляцію через системні механізми ядра та політики доступу. При цьому, якщо зловмисник отримує підвищені права, він потенційно може використовувати інструменти читання пам'яті процесів, наприклад, через інтерфейси дебагу, а також отримати дані, які ОС могла винести зі RAM у swap або інші механізми збереження стану. Тому для Linux доцільні заходи, які поєднують принцип найменших привілеїв, посилену ізоляцію процесів, обмеження дебаг-можливостей та контроль конфігурації swap/шифрування диску, щоб зменшити ризики витоку ключів із RAM. Тому для Linux характерні такі практичні заходи:

- запуск сервісів від окремих користувачів, застосування принципу найменших привілеїв;
- використання механізмів MAC для ізоляції процесів;
- обмеження можливостей дебагу/читання пам'яті процесів, наприклад, контроль ptrace-політик;
- контроль swap/hibernation і шифрування диску, щоб дані з RAM не потрапляли на диск у відкритому вигляді;
- використання апаратних/зовнішніх модулів ключів або софт-сховищ із суворими правами доступу.

Отже, ключова ідея — жорстка ізоляція процесів і контроль механізмів, через які RAM може стати доступною або бути збереженою на диск.

Порівняльна таблиця: загрози для ключів у RAM та методи протидії

Загроза	**Наслідок**	**Методи протидії**	**Особливості/реалізація у Windows**	**Особливості/реалізація у Linux**
Дамп пам'яті процесу (memory dump)	Ключ/проміжні значення можуть потрапит	Мінімізувати час життя ключа в RAM; очищення (zeroization);	Обмеження прав на створення дампів для неадміністраторів; контроль доступу до	Обмеження доступу до механізмів читання пам'яті процесів; політики

	и у файл дампу	ізоляція процесу	інструментів діагностики; розділення ролей/акаунтів	безпеки для дебагу
Crash dump / звіти про падіння	Витік чутливих даних у crash-артефакти	Вимикати/обмежувати crash reporting для чутливих компонентів; не логувати секрети	Налаштування політик збору діагностики; контроль локального доступу до звітів	Контроль core dumps (ulimit), обмеження доступу до файлів дампів
Swap / hibernation (винесення RAM на диск)	Ключі можуть опинитись на диску	Контроль swap/hibernation; шифрування диску	Практично: шифрування диску (щоб артефакти на диску були захищені); контроль конфігурації	Відключення/обмеження swap або шифрування swap; шифрування диску; контроль hibernation
Ін’єкція в процес / перехоплення викликів	Зловмисник може “побачити” ключ у момент використання	Принцип найменших привілеїв; ізоляція (окремий сервіс); контроль цілісності компонентів	Контроль прав; обмеження запуску непідписаних/невідомих компонентів у корпоративному середовищі; ізоляція сервісів	MAC-політики (SELinux/AppArmor), ізоляція сервісів, запуск від окремого користувача
Дебаг (debugger) і читання пам’яті	Прямий доступ до RAM процесу	Обмеження дебагу; мінімальні привілеї; виділення чутливих операцій в окремий процес	Контроль прав на дебаг/діагностику; обмеження доступу до інструментів аналізу процесів	Контроль ptrace та політик доступу; ізоляція процесів

Залишкові дані в heap/stack	Ключ зберігається в пам’яті довше, ніж потрібно	Zeroization; уникати зайвих копій; безпечні структури	Правильна робота з буферами, уникнення логування/рядків; за можливості — робота через системні об’єкти ключів	Аналогічно: безпечне очищення, контроль копій, дисципліна роботи з пам’яттю
Підміна криптобібліотеки (DLL/so hijacking)	Ключ може бути перехоплено “шкідливою” бібліотекою	Контроль залежностей; перевірка джерел/версій; мінімальні привілеї	Контроль шляхів завантаження DLL, політики встановлення ПЗ, контроль підпису/довіри в корпоративному середовищі	Контроль пакетів/репозиторіїв, фіксація версій, контроль прав на каталоги бібліотек
Компрометація прав (локальний admin)	Більшість софт-захистів обходиться	Апаратне сховище ключів (TPM/токен/смарткарта); розділення доступів	TPM/смарткарти/токени, централізовані політики та аудит	HSM/токени через PKCS#11, ізоляція, аудит, MAC-політики

Методи вирішення задачі контролю правильності функціонування програми криптографічної обробки інформації

Правильність функціонування криптографічної програми означає, що вона:

- реалізує алгоритм згідно зі стандартом, специфікацією;
- використовує коректні параметри: довжина ключів, криві, режими, формати;
- забезпечує очікувані властивості безпеки

- коректно обробляє помилки та не допускає небезпечних сценаріїв, наприклад, повторне використання nonce, випадковий генератор низької якості, неправильна перевірка сертифікатів.

На практиці навіть криптографічно стійкий алгоритм може стати небезпечним через помилки реалізації або неправильне використання.

Для підтвердження правильності роботи криптографічної програми доцільно застосовувати сукупність методів тестування та контролю:

1. Тест-вектори (Known Answer Tests, KAT)
   Перевірка результатів на стандартних наборах.
   Це базовий спосіб виявити помилки реалізації алгоритму або параметрів.

2. Unit-тести та regression-тести
   Unit-тести перевіряють окремі функції генерація ключів, підпис, перевірка, шифрування, розшифрування. Regression-тести гарантують, що після змін у коді програма не почала працювати неправильно.

3. Негативні тести (negative testing)
   Мета – перевірити, що система відхиляє некоректні дані: змінений шифротекст або тег автентичності, змінений підпис, некоректні параметри, неправильний формат ключа. Для цифрового підпису, наприклад, зміна 1 байта повідомлення або підпису має призводити до провалу перевірки.

4. Property-based testing
   Тестування властивостей, які мають виконуватися завжди, наприклад, Decrypt(Encrypt(m)) = m для коректних ключів, або те що verify має повертати false, якщо повідомлення змінене.

5. Валідація параметрів та перевірка вхідних даних
   Перед виконанням операцій потрібно перевіряти допустимість параметрів: довжину ключів, приналежність точки кривій, діапазони значень, коректність форматів. Це знижує ризик як випадкових помилок, так і навмисних входів.

Окремою задачею є контроль того, що програма робить все правильно:

- застосовуються рекомендовані довжини ключів і стійкі параметри;
- не використовуються застарілі алгоритми та режими;
- nonce та IV генерується коректно (унікальність т випадковість там, де це потрібно);
- для автентифікованого шифрування перевірка тега є обов’язковою і не може бути пропущена;

- повідомлення про помилки не повинні розкривати чутливі деталі, наприклад, різні помилки на різні причини можуть бути “оракулом”.

Правильні налаштування важливі, бо більшість реальних вразливостей виникає не через поламаний алгоритм, а через некоректну інтеграцію.

Але навіть коректна реалізація може бути скомпрометована, якщо:

- криптографічна бібліотека підмінена, або використано модифіковану залежність;
- зловмисник отримав можливість ін’єкції коду в процес, або дебагу під привілейованим акаунтом;
- збірка виконана з небезпечними прапорами.

Тому до практичних методів контролю відносять:

- фіксацію версій залежностей, перевірку їх цілісності, контроль джерел встановлення;
- підпис/верифікацію компонентів, контроль оновлень;
- запуск чутливих компонентів із мінімальними правами;
- логування подій безпеки без потрапляння чутливих даних у логи.

Порівняння інтерфейсів Crypto API та PKCS#11 у контексті захисту ключів і контролю правильності

Crypto API – це набір системних інтерфейсів і провайдерів, через які прикладні програми виконують криптографічні операції, генерацію ключів, підпис, шифрування, перевірку тощо. Ключова перевага Crypto API полягає в тісній інтеграції з ОС, програма може використовувати стандартні механізми керування ключами, доступами та політиками безпеки, а також отримувати оновлення криптографічних компонентів разом із системними оновленнями.

З точки зору задач цієї лабораторної:

- захист ключів у RAM може покращуватися тим, що програма працює не з ключем у відкритому вигляді, а з системним об’єктом/контейнером ключа, а чутливі операції виконує провайдер;
- контроль правильності підтримується стандартними реалізаціями алгоритмів, валідованими параметрами та типовими механізмами обробки помилок.

Недоліком є прив’язка до конкретної ОС (інтерфейси і поведінка різняться), а також залежність від правильного використання API прикладним розробником.

PKCS#11 – стандартний інтерфейс взаємодії з криптографічними модулями токени, смарткарти, тощо. Головна ідея – ключі зберігаються та використовуються всередині модуля, а застосунок звертається до них через API.

У контексті лабораторної:

- захист ключів у RAM може бути значно кращим, якщо використовується апаратний модуль: приватні ключі можуть бути неекспортованими, а операції підпису та розшифрування виконуються всередині модуля. Тоді секретний ключ взагалі не з’являється в RAM прикладного процесу;
- контроль правильності частково переноситься на модуль – реалізація алгоритмів та обмеження параметрів задаються конкретним провайдером PKCS#11.

Слабке місце – складніша інтеграція, різна якість реалізацій у різних виробників та необхідність коректної роботи з сесіями, параметрами та правами доступу.

Порівняльна таблиця

Вимога	**Crypto API**	**PKCS#11**
Зменшення ризику витоку ключа з RAM	Часто можливий режим роботи через системний контейнер ключа, але багато залежить від реалізації і чи не експортується ключ у відкритому вигляді	Найкращий варіант при апаратному модулі: ключ може бути неекспортованим і не з’являтися в RAM застосунку
Контроль доступу до ключів	Керується засобами ОС (політики, права, інтеграція з акаунтами, сховищами)	Контроль доступу задається політиками токена/HSM (PIN, ролі, ACL), незалежно від ОС
Захист від дампів та дебагу	Частково вирішується політиками ОС, але за компрометації endpoint ключ може опинитись у RAM	Якщо ключ неекспортований і операції в модулі — дамп endpoint не дає ключа (лише може дати дані сесії/виклики)

Контроль правильності криптоопераційй	Стандартні реалізації ОС, часто з перевірками параметрів	Залежить від реалізації вендора; часто модуль нав’язує допустимі алгоритми/параметри
Переносимість між ОС	Низька/середня)	Вища: один стандартний інтерфейс, але потрібні відповідні модулі/драйвери
Складність впровадження	Зазвичай простіше особливо для типових задач на конкретній ОС	Часто складніше: сесії, об’єкти, налаштування токена/модуля, vendor-особливості
Найкращий сценарій застосування	Коли потрібна тісна інтеграція з ОС і швидке впровадження	Коли ключі мають бути апаратно захищені та не виходити за межі модуля

Висновки

У межах лабораторної роботи було розглянуто основні задачі, що виникають при програмній реалізації криптосистем на кінцевих пристроях користувачів. Показано, що практична безпека залежить не лише від стійкості криптоалгоритмів, а й від умов їх виконання в операційній системі: ключова інформація під час роботи може потрапляти в оперативну пам’ять, а також у допоміжні артефакти на кшталт дампів. Тому для зменшення ризиків компрометації ключів у RAM доцільно застосовувати комплекс заходів: мінімізувати час життя ключових даних у пам’яті, виконувати їх коректне очищення після використання, зменшувати кількість копій ключового матеріалу, ізолювати криптографічні операції в окремих компонентах і, за можливості, використовувати апаратні або зовнішні сховища ключів.

Також запропоновано підходи до контролю правильності функціонування криптографічного програмного забезпечення, зокрема застосування тест-векторів, unit/regression-тестів, негативних тестів та перевірок параметрів, а також контроль залежностей і середовища виконання для зниження ризиків підміни бібліотек або втручання в процес. Проведено порівняння інтерфейсів Crypto API та PKCS#11 з точки зору вирішення зазначених задач: Crypto API забезпечує тісну інтеграцію з механізмами ОС і спрощує реалізацію типових сценаріїв, тоді як PKCS#11 є більш придатним у випадках, коли необхідно реалізувати роботу з ключами так, щоб секретний ключовий матеріал не

експортувався у відкритому вигляді в пам’ять прикладного процесу (наприклад, при використанні токенів або HSM), що суттєво підвищує рівень захисту від компрометації ключів на endpoint.