DOI 10.34185/1562-9945-2-157-2025-19
УДК 004.056.53

А.М. Клименко, Н.В. Карпенко, В.В. Герасимов

ШИФРУВАННЯ ТА РОЗШИФРУВАННЯ ДАНИХ У DATASTORE ДЛЯ БЕЗПЕЧНОГО ЛОКАЛЬНОГО ЗБЕРІГАННЯ

Анотація. Актуальність дослідження зумовлена ризиками витоку конфіденційних даних у мобільних додатках. Розглянуто проблему відкритого зберігання даних у DataStore Android. Мета – забезпечити безпечне локальне збереження токенів через AES-шифрування (CBC, PKCS7). Підхід, запропонований у статті, запобігає несанкціонованому доступу, що підвищує безпеку чутливої інформації
Ключові слова: шифрування, DataStore, AES, безпека даних, Android, токени, конфіденційність.

Постановка проблеми. Локальне зберігання даних на мобільних пристроях є важливою складовою безпеки персональної та корпоративної інформації. Смартфони та планшети часто містять конфіденційні дані, такі як паролі, фінансові відомості, приватні повідомлення та робочі документи. Якщо ці дані недостатньо захищені, вони можуть стати ціллю для зловмисників, що загрожує як особистій, так і професійній безпеці користувача.

Основними ризиками локального зберігання є фізичний доступ до пристрою, злом операційної системи та витік даних через шкідливі програми. Недостатній рівень шифрування або слабкі паролі можуть зробити пристрій вразливим до атак. Тому сучасні операційні системи мобільних пристроїв пропонують такі засоби безпеки, як апаратне шифрування, біометричну автентифікацію та захищені середовища для збереження критичних даних.

Аналіз останніх досліджень і публікацій. Android пропонує кілька варіантів зберігання даних:

- ✓ локальне сховище (Internal Storage) - доступне лише для самого додатка;
- ✓ зовнішнє сховище (External Storage) - раніше SD-карта, зараз - це спільне сховище на пристрої;
- ✓ спільні налаштування (SharedPreferences) - для зберігання пари ключ-значення;
- ✓ SQLite-база даних - для збереження структурованих даних;
- ✓ Keystore - для зберігання криптографічних ключів;
- ✓ хмарне сховище - дані можуть зберігатися на сервері.

У статті [1] запропоновано використання динамічного аналізу, який дозволяє виявити небезпечне зберігання даних у реальному часі, без потреби доступу до вихідного коду додатка. Автори цієї статті вказують на те, що Scoped Storage суттєво покращує

безпеку, але багато додатків обходяться без нього. Окрім цього, деякі додатки мають небезпечний доступ до системних файлів та зберігають чутливі дані у спільних папках, що створює ризики витоку інформації. Але у статті не розглянуто, чи використовують додатки шифрування для захисту чутливих даних, а також те, чи перевіряються ключі доступу до Keystore.

Автори статті [2] пропонують наступні рішення для захисту даних:

- фізичні рішення: CleanOS, TinMan, Sentry, Deadbolt, DroidVault та ARMOR (захист від RowHammer-атак);
- програмні рішення: використання Android Encryption Systems (FDE та KeyChain), біометричної автентифікації та хмарних технологій.
- вирішення проблем з впровадженням криптографічних API через помилки розробників.

Однак вплив шифрування на швидкодію додатків та автономність пристрою залишається поза їх увагою. Окрім цього, не розглянуть питання: «Чи використовують додатки належне шифрування?», «Чи достатньо захищений Keystore від компрометації?»

У статті [3] обговорюється важливість безпеки даних у сучасному цифровому світі та представлений інструмент для захисту чутливої інформації HashiCorp Vault. Він може шифрувати та розшифровувати дані без їх зберігання, що корисно для додатків, які потребують можливості шифрування без видачі ключів шифрування. Однією з його можливостей є безпечне управління обліковими даними баз даних, ключами API та SSH-сертифікатами. Однак стаття не містить практичних прикладів або сценаріїв впровадження HashiCorp Vault, що може ускладнити розуміння його застосування, а огляд функцій Vault є загальним і не заглиблюється в деталі реалізації або налаштування.

Деякі особливості налаштування Vault висвітлюються у [4], хоча автори також кажуть про те, що правильно налаштувати Vault для проектів зі складною структурою непросто і доведеться витратити чимало часу на правильне структурування і налаштування доступів та динамічних паролей.

Мета дослідження. Дослідити проблему відкритого зберігання даних у сховищі Data Store Android-додатків та запропонувати метод безпечного зберігання чутливої інформації на прикладі токенів доступу.

Викладення основного матеріалу дослідження. У сучасних Android-додатках активно використовують локальне зберігання даних. Data Store є сучасним та ефективним рішенням для цього, оскільки зберігає локальні дані у вигляді пари ключ-значення. Однак Data Store зберігає дані у відкритому вигляді, тобто користувач або шкідливе програмне забезпечення можуть їх зчитати.

Якщо зберігається нечутлива інформація, наприклад, чи хоче користувач отримувати повідомлення або чи увімкнений темний режим екрану, — це не є проблемою. Проте, якщо ми зберігатимемо токен, за допомогою якого користувач виконуватиме запити до API, можуть виникнути проблеми. Зловмисне програмне забезпечення може використовувати цей токен для надсилання запитів до сервера від імені користувача.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Це може призвести до додаткових витрат на обслуговування сервера та створити ризики витоку конфіденційних даних користувача.

Для шифрування даних на мовах Java/Kotlin є пакет javax.crypto. Цей пакет надає класи та інтерфейси для криптографічних операцій. Криптографічні операції, визначені у цьому пакеті, включають шифрування, генерацію ключів і узгодження ключів, а також генерацію коду автентифікації повідомлень (MAC). Підтримка шифрування включає симетричні, асиметричні, блокові та потокові шифри. Цей пакет також підтримує захищені потоки і запечатані об'єкти [5].

Jetpack DataStore — це рішення для зберігання даних, яке дозволяє зберігати пари ключ-значення або типізовані об'єкти за допомогою буферів протоколу. DataStore використовує Kotlin-корутини та Flow для асинхронного, послідовного та транзакційного зберігання даних [6].

Для шифрування даних було обрано симетричний блочний алгоритм AES, однією з перевагою якого є змінна довжина ключа. Симетричні ключі можуть мати довжину 128, 192 або 296 біт. Чим довший ключ, тим складніше його зламати [7].

Алгоритм AES використовує Cipher Block Chaining (CBC) та PKCS7 Padding, оскільки він працює з блоками фіксованого розміру і потребує заповнення останнього блоку перед шифруванням.

Режим CBC — це типовий режим роботи алгоритму блочного шифрування, який забезпечує можливість обробки даних за стандартом шифрування даних (DES) і розширеним стандартом шифрування (AES), при чому довжина шифру для DES повинна становити 64 біти, а для AES відповідно 128/192/256 біт. Потрібно звернути увагу на те, що в цьому режимі алгоритм працює з блоками фіксованого розміру (64 або 128 біт для одного блоку), але в реальному світі текст має різну довжину. Тому останній блок тексту, що шифрується або дешифрується, повинен бути збільшений до 128 біт [8].

Завдяки використанню режиму CBC кожен блок шифротексту залежить не тільки від відповідного відкритого тексту, а й від попереднього зашифрованого блоку. Це робить шифрування стійким до повторень однакових блоків та гарантує, що навіть невелика зміна у відкритому тексті призведе до кардинальної зміни у вихідному шифротексті. Оскільки CBC використовує ініціалізаційний вектор (iv), він додає певну випадковість до процесу шифрування і запобігає атакам на повторювані шаблони в даних.

Доповнення даних (Padding) застосовується перед шифруванням, коли це ключове слово вказується за допомогою виклику служби Symmetric Algorithm Encipher, і видаляється з розшифрованих даних, коли це ключове слово вказується за допомогою виклику служби Symmetric Algorithm Decipher [9]. Тобто перед шифруванням текст "вирівнюється" по межі в 128 біт, а після розшифрування цей "вирівнюючий" текст видаляється з вже розшифрованих даних.

Правила заповнення PKCS дуже прості:

1. Байти заповнення завжди додаються до відкритого тексту перед його шифруванням.

2. Кожен байт заповнення має значення, що дорівнює загальній кількості байт заповнення, які додаються. Наприклад, якщо потрібно додати 6 байт вирівнювання, кожен з цих байтів матиме значення 0x06.

3. Загальна кількість байтів заповнення принаймні дорівнює одиниці, і це кількість, яка необхідна для того, щоб довести довжину даних до розміру, кратного розміру блоку алгоритму шифрування [9].

Щоб реалізувати шифрування та розшифрування, потрібно спочатку створити константи, в яких задаємо алгоритм шифрування, режим блочного шифрування, схему доповнення, щоб дані відповідали вимогам алгоритму, а також поле, яке об'єднує ці параметри, та ім'я ключа, під яким він буде зберігатися.

Приклад констант:

```
private const val KEY_ALIAS = "secret"
private const val ALGORITHM = KeyProperties.KEY_ALGORITHM_AES
private const val BLOCK_MODE = KeyProperties.BLOCK_MODE_CBC
private const val PADDING = KeyProperties.ENCRYPTION_PADDING_PKCS7
private const val TRANSFORMATION = "$ALGORITHM/$BLOCK_MODE/$PADDING"
```

Далі потрібно ініціалізувати cipher (об'єкт для шифрування/розшифрування, який використовує попередньо визначену трансформацію) та keyStore (об'єкт, який працює з AndroidKeyStore — місцем для зберігання ключів, забезпечуючи їхню безпеку).

Приклад ініціалізації:

```
private val cipher = Cipher.getInstance(TRANSFORMATION)
private val keyStore = KeyStore.getInstance("AndroidKeyStore").apply
{ load(null) }
```

Оскільки доступ до даних відбувається через ключ, потрібно створити методи для його отримання або створення.

```
    private fun getKey(): SecretKey {
    val existingKey = keyStore.getEntry(KEY_ALIAS, null) as? Key-
Store.SecretKeyEntry
    return existingKey?.secretKey ?: createKey()}

private fun createKey(): SecretKey {
    return KeyGenerator
        .getInstance(ALGORITHM)
        .apply {
            init(
                KeyGenParameterSpec.Builder(
                    KEY_ALIAS,
                    KeyProperties.PURPOSE_ENCRYPT or
                    KeyProperties.PURPOSE_DECRYPT)
                    .setBlockModes(BLOCK_MODE)
                    .setEncryptionPaddings(PADDING)
                    .setRandomizedEncryptionRequired(true)
                    .build()
```


ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

```
            )}
        .generateKey()}
```

BLOCK_MODE та PADDING — для встановлення блочного режиму та доповнення. Останній крок — це створення методів шифрування та розшифрування. Приклад методів:

```
    fun encrypt(bytes: ByteArray): ByteArray {
    // Ініціалізація шифрувальника в режимі ENCRYPT_MODE
cipher.init(Cipher.ENCRYPT_MODE, getKey())
    // Отримання ініціалізаційного вектора (iv)
val iv = cipher.iv
    // Шифрування вхідних байтів
val encrypted = cipher.doFinal(bytes)
    // Повернення iv разом із зашифрованими даними
return iv + encrypted
    }

    fun decrypt(bytes: ByteArray): ByteArray {
    // Витягуємо iv із зашифрованих даних
val iv = bytes.copyOfRange(0, cipher.blockSize)
    // Отримуємо зашифровану частину
    val data = bytes.copyOfRange(cipher.blockSize, bytes.size)
    // Ініціалізація розшифрувальника
    cipher.init(Cipher.DECRYPT_MODE, getKey(), IvParameterSpec(iv))
    return cipher.doFinal(data) // Розшифрування даних
    }
    Методи encrypt і decrypt використовуються в UserPreferencesSeri-
alizer для шифрування та розшифрування даних перед їх збереженням у
Data Store.
    @Serializable
data class UserPreferences(
    val token: String? = null
)

object UserPreferencesSerializer : Serializer<UserPreferences> {
    override val defaultValue: UserPreferences
        get() = UserPreferences()

    override suspend fun readFrom(input: InputStream): UserPrefer-
ences {
        val encryptedBytes = withContext(Dispatchers.IO) {
            input.use { it.readBytes() }}
        val encryptedBytesDecoded =
Base64.getDecoder().decode(encryptedBytes)
        val decryptedBytes = Crypto.decrypt(encryptedBytesDecoded)
        val decodedJsonString = decryptedBytes.decodeToString()
        return Json.decodeFromString(decodedJsonString) }
```

```
    override suspend fun writeTo(t: UserPreferences, output: Out-
putStream) {
        val json = Json.encodeToString(t)
        val bytes = json.toByteArray()
        val encryptedBytes = Crypto.encrypt(bytes)
        val encryptedBytesBase64 =
Base64.getEncoder().encode(encryptedBytes)
        withContext(Dispatchers.IO) {
            output.use {
                it.write(encryptedBytesBase64)
            }}}}
```

Метод **readFrom(input: InputStream)** відповідає за зчитування збережених у DataStore даних, їх декодування та перетворення у об'єкт UserPreferences. Спочатку він читає вхідні зашифровані байти та декодує їх з рядку в кодуванні Base64. Далі отримані дані розшифровуються за допомогою Crypto.decrypt(), після чого отримані байти перетворюються у JSON-рядок. Нарешті, цей рядок десеріалізується у UserPreferences за допомогою Json.decodeFromString(), і отриманий об'єкт повертається.

Метод **writeTo(t: UserPreferences, output: OutputStream)** виконує зворотну операцію - бере об'єкт UserPreferences, перетворює його у JSON-рядок та шифрує перед збереженням у DataStore. Спочатку об'єкт серіалізується у JSON через Json.encodeToString(t), потім конвертується у масив байтів. Отримані байти шифруються за допомогою Crypto.encrypt(), після чого байти даних кодуються в рядок в кодуванні Base64, який записується до output, щоб бути збереженим у DataStore (рис. 1).

Для демонстрації працездатності коду створимо інтерфейс із двома кнопками та текстовим полем, яке виводитиме декодовані дані з Data Store (рис. 2).

```
    private val Context.dataStore by dataStore(
        fileName = "user-preferences",
        serializer = UserPreferencesSerializer
    )

    private val token = "some token"

    class MainActivity : ComponentActivity() {
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            enableEdgeToEdge()
            setContent {
                EncryptedDataStoreTheme {
                    Scaffold(modifier = Modifier.fillMaxSize()) {
innerPadding ->
                        Column(
                            modifier = Modifier
                                .padding(innerPadding)
                        ) {
```


ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

```
                    val scope = rememberCoroutineScope()
                    var text by remember {mutableS-
tateOf("")}
                    Button(
                        onClick = {
                            scope.launch {
                                dataStore.updateData {
                                    UserPreferences(
                                        token = token
                                    ) }}}
                    ) {
                        Text("Encrypt")
                    }
                    Button(
                        onClick = {
                            scope.launch {
                                text = dataS-
tore.data.first().token ?: ""
                            }}
                    ) {
                        Text("Decrypt")
                    }
                    Text(
                        text = text
                    ) }}}}}}
```

Тепер у змінній token зберігатимемо рядок “some token”.

```
gZdzt5TZDCYl9+FNdYB+Gk2fixlP1TAmR6TFa6Uzpi2uhDUdxUeWtFdIcGUl7piU
```

Рисунок 1 – Вигляд шифрованого тексту у Data Store файлі

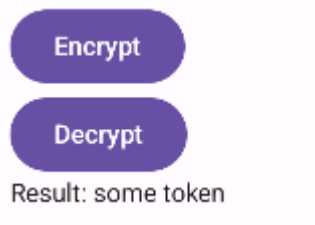

Рисунок 2 - Вигляд розшифрованого тексту на екрані мобільного додатку

Висновки. Шифрування даних у Android-додатках є важливим елементом безпечного локального зберігання інформації. Використання Data Store для збереження чутливих даних, таких як токени, вимагає застосування шифрування для захисту їх від несанкціонованого доступу. Рішення, яке передбачає використання AES алгоритму з режимом CBC та схемою доповнення PKCS7, дозволяє забезпечити належний рівень безпеки та запобігти потенційним загрозам з боку зловмисників.

ЛІТЕРАТУРА

1. Kris Heid, Tobias Tefke, Jens Heider and Ralf C. Staudemeyer. Android Data Storage Locations and What App Developers Do with It from a Security and Privacy Perspective. / Proceedings of the 8th International Conference on Information Systems Security and Privacy ICISSP. – 2022. – Volume 1. – pp. 378-387. DOI: 10.5220/0010838200003120

2. Haya Altuwaijri, Sanaa Ghouzali. Android data storage security: A review. / Journal of King Saud University - Computer and Information Sciences. – 2018. – 32(5). DOI: 10.1016/j.jksuci.2018.07.004

3. Myroslav Kyselytsia. Захист ваших секретів: Глибоке дослідження HashiCorp Vault. [Електронний ресурс]. Режим доступу: https://dou.ua/forums/topic/46007/

4. Скок М., Герасимов В. Технологія Vault від HashiCorp для зберігання та захисту паролів та токенів доступу. / VII Всеукраїнська науково-практична конференція "Перспективні напрямки сучасної електроніки, інформаційних і комп'ютерних систем" (MEICS-2022), м. Дніпро, ДНУ ім. О. Гончара, 23-25 листопада 2022 р., с. 66-67. http://meics.dnure.dp.ua/files/MEICS-2022.pdf

5. Package javax.crypto [Електронний ресурс]. Режим доступу: https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html

6. DataStore [Електронний ресурс]. Режим доступу: https://developer.android.com/topic/libraries/architecture/datastore

7. What is AES Encryption and why is it important? [Електронний ресурс]. Режим доступу: https://winzip.com/blog/enterprise/aes-encryption-explained/?srsltid=AfmBOorBsgzMt91e9tSB1cHpKOi8taud8u7IpmniFkhiDKHMgQ1D304I

8. CBC Mode [Електронний ресурс]. Режим доступу: https://xilinx.github.io/Vitis_Libraries/security/2020.1/guide_L1/internals/cbc.html#overview

9. PKCS padding method [Електронний ресурс]. Режим доступу: https://www.ibm.com/docs/en/zos/3.1.0?topic=rules-pkcs-padding-method

REFERENCES

1. Kris Heid, Tobias Tefke, Jens Heider and Ralf C. Staudemeyer. Android Data Storage Locations and What App Developers Do with It from a Security and Privacy Perspective. / Proceedings of the 8th International Conference on Information Systems Security and Privacy ICISSP. – 2022. – Volume 1. – pp. 378-387. DOI: 10.5220/0010838200003120

2. Haya Altuwaijri, Sanaa Ghouzali. Android data storage security: A review. / Journal of King Saud University - Computer and Information Sciences. – 2018. – 32(5). DOI: 10.1016/j.jksuci.2018.07.004

3. Myroslav Kyselytsia. Zakhyst vashykh sekretiv: Hlyboke doslidzhennia HashiCorp Vault. https://dou.ua/forums/topic/46007/

4. Skok M., Gerasymov V. Tekhnolohiia Vault vid HashiCorp dlia zberihannia ta zakhystu paroliv ta tokeniv dostupu. / VII Vseukrainska naukovo-praktychna konferentsiia "Perspektyvni napriamky su-chasnoi elektroniky, informatsiinykh i kompiuternykh system" (MEICS-2022), m. Dnipro, DNU im. O. Honchara, 23-25 november 2022. – p. 66-67. http://meics.dnure.dp.ua/files/MEICS-2022.pdf

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

5. Package javax.crypto https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html
6. DataStore https://developer.android.com/topic/libraries/architecture/datastore
7. What is AES Encryption and why is it important? https://winzip.com/blog/enterprise/aes-encryption-explained/?srsltid=AfmBOorBsgzMt91e9tSB1cHpKOi8taud8u7IpmniFkhiDKHMgQ1D304I
8. CBC Mode https://xilinx.github.io/Vitis_Libraries/security/2020.1/guide_L1/internals/cbc.html#overview
9. PKCS padding method https://www.ibm.com/docs/en/zos/3.1.0?topic=rules-pkcs-padding-method

Received 26.03.2025.
Accepted 28.03.2025.

Encryption and Decryption of Data in DataStore for Secure Local Storage

The relevance of this study is determined by the growing threats of data leakage in mobile applications, where sensitive user information such as tokens, passwords, and API keys are often stored insecurely. Local storage on Android devices remains a critical aspect of application security, as improper handling of sensitive data can lead to unauthorized access, data breaches, and financial or reputational damage. This research focuses on addressing the problem of insecure storage within Android's DataStore and proposes a secure encryption-based approach to mitigate risks.

The problem statement highlights that DataStore, a modern and efficient key-value storage solution for Android applications, lacks built-in encryption mechanisms, leaving sensitive information vulnerable to unauthorized access. While DataStore provides an efficient and structured way to store small amounts of persistent data asynchronously, its default implementation does not offer protection against potential data exposure in case of device compromise or malware attacks. This raises the need for additional security measures to ensure that confidential information, such as authentication tokens, remains protected.

The objective of this research is to develop and implement a secure method for encrypting sensitive data stored in DataStore. The goal is to integrate an effective encryption mechanism that enhances data security without compromising performance or usability in Android applications. The study focuses on AES encryption (Advanced Encryption Standard) using Cipher Block Chaining (CBC) mode with PKCS7 padding to ensure strong protection against unauthorized access. AES encryption is widely recognized for its robustness, and the chosen configuration enhances security by introducing randomization and integrity checks.

The methodology involves an in-depth analysis of existing Android storage mechanisms, a comparative evaluation of encryption techniques, and the development of an encryption layer integrated with DataStore. The proposed encryption scheme was implemented using Javax.crypto libraries, ensuring compatibility with modern Android security best practices. The research also examines the impact of encryption on application performance, assessing factors such as processing time, storage efficiency, and integration complexity.

The results demonstrate that incorporating AES encryption significantly improves the security of sensitive data in DataStore without introducing substantial performance overhead. The experimental implementation confirms that encrypted tokens stored in DataStore remain

protected from unauthorized extraction, even in cases where an attacker gains access to the file system. Additionally, the research highlights the importance of proper key management, advocating the use of Android Keystore for securely generating and storing encryption keys.

Key conclusions drawn from the study emphasize that encryption is essential for preventing unauthorized access to sensitive data stored in DataStore. The implementation of AES-CBC encryption with PKCS7 padding effectively enhances security while maintaining efficiency. Future research directions include exploring more advanced encryption modes, such as AES-GCM, and evaluating their performance trade-offs in real-world applications. The study also recommends further research on user authentication mechanisms to complement data encryption and strengthen overall application security.

Keywords: encryption, DataStore, AES, data security, Android, tokens, privacy.

Клименко Артем Максимович – студент 4 курсу кафедри електронних обчислювальних машин ДНУ ім. Олеся Гончара, м. Дніпро.
Карпенко Надія Валеріївна – к.ф.-м.н., доцент, доцент кафедри електронних обчислювальних машин ДНУ ім. Олеся Гончара.
Герасимов Володимир Володимирович – к.т.н., доцент, завідувач кафедри комп'ютерних наук та інформаційних технологій ДНУ ім. Олеся Гончара.

Klymenko Artem – 4th year student of the Department of Electronic Computing, Oles Honchar Dnipro National University, Dnipro.
Karpenko Nadiya – Candidate of Physical and Mathematical Sciences, Associate Professor of the Department of Electronic Computing, Oles Honchar Dnipro National University, Dnipro.
Gerasimov Volodymyr – Candidate of Technical Sciences, Associate Professor, Head of the Department of Computer Science and Information Technologies, Oles Honchar Dnipro National University, Dnipro.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)