МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія (кафедра) комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

АНАЛІЗ АРХІТЕКТУРИ БЕЗПЕКИ ВЕЛИКИХ ПРОГРАМНИХ СИСТЕМ

Виконав: студент групи 1П-21
Спеціальності
121 Інженерія програмного забезпечення
Микола ГОНЧАРЕНКО
Керівник:
Майя ЛЮТА

Черкаси 2025

АНОТАЦІЯ

Кваліфікаційна робота на тему «Аналіз архітектури безпеки великих програмних систем» складається з вступу, основної частини, що містить 4 розділи, висновку та списку використаних джерел. Загальний обсяг роботи – 73 сторінки. У роботі 8 рисунків та 25 таблиць. Перелік використаних ресурсів налічує 40 одиниць.

У даній роботі здійснено ґрунтовний аналіз сучасних методів захисту інформації від несанкціонованого доступу. Розглянуто основні загрози, що виникають у процесі зберігання, передавання та обробки даних, а також засоби їх нейтралізації. Особливу увагу приділено технічним та програмним способам забезпечення конфіденційності, цілісності та доступності інформації.

Проведено аналіз програмного забезпечення, що використовується для реалізації криптографічного захисту. Оцінено їх функціональні можливості, рівень захисту, зручність використання та відповідність міжнародним стандартам інформаційної безпеки.

Завдяки комплексному підходу в роботі висвітлено як теоретичні основи, так і практичні аспекти організації захисту даних, що дозволяє сформувати цілісне уявлення про актуальні методи інформаційної безпеки та їх ефективне застосування в реальних умовах.

Ключові слова: ЗАХИСТ ДАНИХ, ЕЛЕКТРОННА ЗВІТНІСТЬ, НЕСАНКЦІОНОВАНИЙ ДОСТУП, ІНФОРМАЦІЙНА БЕЗПЕКА, УПРАВЛІННЯ ДОСТУПОМ, АУТЕНТИФІКАЦІЯ, ЗАХИСТ КОНФІДЕНЦІЙНОСТІ, СИСТЕМИ КОНТРОЛЮ ДОСТУПУ.

ANNOTATION

The qualification work on the topic "Analysis of the security architecture of large software systems" consists of an introduction, the main part, which contains 4 sections, a conclusion and a list of sources used. The total volume of the work is 73 pages. The work contains 8 figures and 25 tables. The list of resources used has 40 units.

This work provides a thorough analysis of modern methods of protecting information from unauthorized access. The main threats that arise in the process of storing, transmitting and processing data, as well as means of neutralizing them, are considered. Special attention is paid to technical and software methods for ensuring the confidentiality, integrity and availability of information.

The analysis of software used to implement cryptographic protection is carried out. Their functionality, level of protection, ease of use and compliance with international information security standards are assessed.

Thanks to a comprehensive approach, the work highlights both the theoretical foundations and practical aspects of data protection organization, which allows you to form a holistic view of current information security methods and their effective application in real conditions.

Keywords: DATA PROTECTION, ELECTRONIC REPORTING, UNAUTHORIZED ACCESS, INFORMATION SECURITY, ACCESS MANAGEMENT, AUTHENTICATION, PRIVACY PROTECTION, ACCESS CONTROL SYSTEMS.

ЗМІСТ

ВСТУП

У сучасному цифровому світі безпека програмних систем набуває все більшого значення, адже саме ці системи є основою функціонування практично всіх сфер суспільного життя – від економіки та охорони здоров’я до державного управління та оборони. Із стрімким розвитком технологій, збільшенням кількості кіберзагроз, зростанням складності архітектур програмного забезпечення та їх розгортанням у хмарному середовищі, виникає нагальна потреба у впровадженні ефективних стратегій захисту. Великі програмні системи, що складаються з десятків або навіть сотень взаємопов’язаних модулів, сервісів і баз даних, мають унікальні особливості з точки зору побудови захисту, оскільки в них надзвичайно високим є ризик появи критичних вразливостей, а також масштаб наслідків від потенційних інцидентів.

Актуальність теми дослідження зумовлена тим, що на практиці безпека таких систем часто ускладнена відсутністю єдиної політики контролю доступу, фрагментованістю засобів моніторингу, використанням застарілих компонентів та недостатньою інтеграцією безпеки у процеси розробки. З огляду на стрімке зростання обсягів даних, розповсюдження концепцій DevOps, мікросервісів, контейнеризації та активне впровадження хмарних технологій, питання надійного захисту системного ландшафту постає як першочергове завдання для підприємств і організацій, що прагнуть забезпечити конфіденційність, цілісність та доступність своїх інформаційних активів.

Об’єктом дослідження є великі програмні системи з розподіленою архітектурою, які функціонують у корпоративному або хмарному середовищі.

Предметом дослідження є методи, принципи, інструменти та архітектурні рішення щодо забезпечення безпеки в таких програмних системах, а також виявлення потенційних вразливостей і механізмів реагування на інциденти.

Метою дослідження є аналіз архітектури безпеки великих програмних систем, виявлення її слабких місць, систематизація сучасних підходів до управління доступом, моніторингу, шифрування, виявлення загроз, а також формування практичних рекомендацій щодо удосконалення захисту таких систем відповідно до сучасних викликів.

Завдання дослідження:

1. Проаналізувати основні принципи побудови архітектури безпеки великих програмних систем та оцінити їх ефективність у сучасних умовах.
2. Охарактеризувати базові механізми захисту, такі як автентифікація, авторизація, шифрування, аудит та моніторинг.
3. Провести аналіз типових загроз, пов’язаних із використанням вразливого коду, соціальною інженерією, шкідливим ПЗ та внутрішніми порушеннями.
4. Дослідити моделі управління ідентифікацією та доступом (IAM), підходи до побудови Zero Trust та автоматизацію реагування на інциденти.
5. Розробити рекомендації щодо вдосконалення архітектури безпеки з урахуванням вимог масштабованості, надійності та відповідності сучасним стандартам безпеки.

Методи дослідження, які використовуються в роботі, включають системний аналіз, моделювання архітектури програмних систем, аналітичний огляд сучасних технічних стандартів і практик (OWASP, NIST, CIS), а також порівняльний аналіз існуючих рішень у сфері кібербезпеки.

Наукова новизна роботи полягає у комплексному узагальненні підходів до проектування архітектури безпеки у масштабних програмних системах, а також у запропонованій моделі автоматизованого контролю доступу, моніторингу та реагування на інциденти, яка враховує специфіку сучасного середовища розгортання програмного забезпечення (контейнеризація, CI/CD, гібридна хмара).

Практичне значення роботи полягає у можливості використання результатів дослідження для підвищення рівня захищеності підприємств, що

експлуатують великі програмні системи, у побудові або оптимізації внутрішніх політик безпеки, розгортанні SIEM, PAM, IAM-систем, а також у процесі підготовки до сертифікацій відповідності вимогам ISO/IEC 27001.

Таким чином, дослідження має як теоретичну, так і прикладну цінність, оскільки дозволяє систематизувати знання з архітектури кіберзахисту та запропонувати практичні шляхи її вдосконалення в умовах зростання цифрових ризиків.

РОЗДІЛ 1
ОГЛЯД ОСНОВНИХ ПРИНЦИПІВ АРХІТЕКТУРИ БЕЗПЕКИ ВЕЛИКИХ ПРОГРАМНИХ СИСТЕМ

Принцип мінімізації прав доступу

У контексті архітектури безпеки великих програмних систем принцип мінімізації прав доступу відіграє одну з ключових ролей у забезпеченні стійкості до внутрішніх та зовнішніх загроз. Цей принцип передбачає, що кожному користувачеві, процесу або компоненту системи мають надаватися лише ті права доступу, які є строго необхідними для виконання конкретних функцій або завдань. Іншими словами, доступ повинен бути обмежений до мінімального рівня, необхідного для нормального функціонування суб'єкта в рамках системи. Такий підхід дозволяє зменшити площу потенційної атаки і обмежити можливі наслідки у разі компрометації облікового запису або програмного компонента.

Механізм мінімізації прав доступу реалізується через встановлення чітких політик доступу, які визначають, які дії дозволено виконувати кожному суб'єкту системи. Ці політики мають бути налаштовані таким чином, щоб не допускати надмірних або необґрунтованих привілеїв. Наприклад, працівнику, який займається виключно обробкою клієнтських запитів, не повинен надаватися доступ до адміністративних інструментів або конфіденційних даних компанії. У великих програмних системах, де кількість користувачів може досягати тисяч, автоматизація управління правами доступу за допомогою спеціалізованих систем управління ідентифікацією та доступом (IAM) є обов'язковою умовою для підтримки безпеки [12].

Реалізація цього принципу також охоплює сегментування доступу в межах самих програмних компонентів. Наприклад, мікросервіси повинні взаємодіяти лише з тими сервісами та API, які необхідні для виконання їхніх функцій. Надання кожному сервісу унікальних ключів доступу або токенів з

обмеженим терміном дії дозволяє знизити ризик ескалації привілеїв у разі зламу. У цьому контексті важливим є також принцип «розділення обов'язків» (Separation of Duties), який є розширенням принципу мінімізації і передбачає, що жоден користувач або система не повинні мати повного контролю над критичним процесом без участі інших сторін.

Одним із практичних інструментів реалізації цього принципу є рольова модель доступу (RBAC – Role–Based Access Control), у якій доступні дії обмежуються на основі ролей, що закріплюються за користувачами. Наприклад, у великій програмній системі може бути визначено ролі «аналітик», «тестувальник», «адміністратор», «розробник», кожна з яких має строго регламентовані привілеї. Розширена модель – атрибутно-орієнтований контроль доступу (ABAC) – дозволяє враховувати додаткові параметри, зокрема час доступу, тип пристрою, місцезнаходження тощо. Застосування таких моделей у масштабних системах дозволяє забезпечити динамічну адаптацію політик доступу до умов роботи.

Розгортання багаторівневих систем контролю доступу також є важливим аспектом реалізації принципу мінімізації. Зокрема, слід впроваджувати механізми двофакторної аутентифікації (2FA), контролю за використанням службових облікових записів, журналювання спроб доступу та інші засоби, які дозволяють не лише обмежити доступ, а й виявляти та реагувати на спроби несанкціонованого використання привілеїв. Також важливо, щоб усі дії, пов'язані з наданням, зміною або відкликанням доступу, були належно задокументовані та підлягали перевірці.

Зі зростанням складності програмних систем та появою нових форм загроз питання мінімізації прав доступу набуває ще більшого значення. Наприклад, зростає популярність підходу Zero Trust, який базується на припущенні, що жоден користувач або пристрій не є довіреним за замовчуванням. У межах цієї концепції доступ надається виключно на підставі постійної верифікації, а кожна спроба отримати доступ оцінюється в режимі реального часу з урахуванням численних факторів ризику.

Отже, принцип мінімізації прав доступу є фундаментальним елементом побудови безпечної архітектури великих програмних систем. Його реалізація дозволяє не лише запобігти багатьом типам атак, але й локалізувати їх наслідки у випадку порушення безпеки. Ретельно розроблена система прав доступу, підтримана технічними засобами контролю та аудиту, є запорукою збереження конфіденційності, цілісності та доступності даних в умовах складних інформаційних екосистем [28].

Принцип безпеки за замовчуванням

Принцип безпеки за замовчуванням (Security by Default) передбачає, що будь–яка система або її компонент повинні бути сконфігуровані з урахуванням максимальної безпеки з самого початку, ще до першого використання або взаємодії з користувачем. У великих програмних системах, де кількість модулів, користувачів і точок доступу є надзвичайно великою, забезпечення безпечного функціонування "за замовчуванням" суттєво знижує ризики пов’язані з людським фактором, некоректними налаштуваннями або недоліками інтеграції. У рамках цього принципу всі сервіси, порти, привілеї або функції, які не є критично необхідними для основного функціоналу, повинні бути відключені або заблоковані до моменту явного дозволу.

Безпека за замовчуванням має вирішальне значення у хмарних інфраструктурах, мультиорендних системах, мікросервісних архітектурах та складних CI/CD–конвеєрах. Адже саме в цих середовищах дрібні недопрацювання або відкриті налаштування можуть стати причиною масштабного витоку інформації. Наприклад, якщо за замовчуванням новостворений обліковий запис адміністратора має необмежений доступ до всіх систем і не потребує додаткової верифікації, то це відкриває широкі можливості для зловмисників у разі компрометації облікових даних. Відповідно, початкова конфігурація має передбачати обмеження, які

користувач може зняти тільки після виконання чітко регламентованих дій, як от підтвердження особи, активація MFA, реєстрація пристрою тощо.

Впровадження принципу безпеки за замовчуванням вимагає тісної співпраці між архітекторами програмного забезпечення, DevOps–командами та фахівцями з кібербезпеки на всіх етапах життєвого циклу системи – від проектування до розгортання та супроводу. Особливо важливим є створення безпечного середовища розробки, в якому жоден код не буде введено в експлуатацію без попередньої перевірки на відповідність політикам безпеки. Наприклад, впровадження обов'язкового статичного аналізу коду, автоматичне відключення вразливих бібліотек або заборона запуску сервісів з відкритими портами без SSL можуть значно покращити безпеку системи ще до її публічного запуску.

Іншою складовою реалізації принципу є формування надійних шаблонів конфігурації (secure configuration templates), які можна повторно використовувати у різних середовищах без необхідності ручного налаштування. Це стосується як налаштування серверів, БД та API, так і стандартів мережевої безпеки, таких як політики міжмережевого екрану, контроль вхідного/вихідного трафіку, параметри шифрування тощо. Наприклад, конфігураційний шаблон для бази даних може включати заборону на виконання зовнішніх запитів, шифрування таблиць, відключення адміністративних функцій без 2FA тощо.

Принцип безпеки за замовчуванням також передбачає налаштування звітності й аудиту за замовчуванням. Усі системи повинні мати активоване журналювання критичних подій, спроб несанкціонованого доступу, змін у конфігурації та збоїв. Крім того, слід забезпечити інтерфейс для регулярного аналізу цих журналів, щоб своєчасно виявляти індикатори компрометації або порушення політик. У складних програмних системах для цього часто використовуються спеціалізовані рішення – SIEM-системи (Security Information and Event Management), які збирають та аналізують дані безпеки з багатьох джерел у реальному часі [7].

Загалом реалізація принципу безпеки за замовчуванням є ефективним способом зниження ризиків ще до того, як система починає взаємодіяти з потенційно ненадійним середовищем. Його дотримання дозволяє уникнути ситуацій, коли користувачі або адміністратори несвідомо активують функції, що відкривають нові вектори атак. Як наслідок, програмна система стає менш вразливою до типових методів зламу, зокрема атаки через незахищені конфігурації, відкриті порти, застарілі версії інструментів тощо. Таким чином, принцип безпеки за замовчуванням є не лише технічною вимогою, а й частиною філософії розробки безпечного програмного забезпечення.

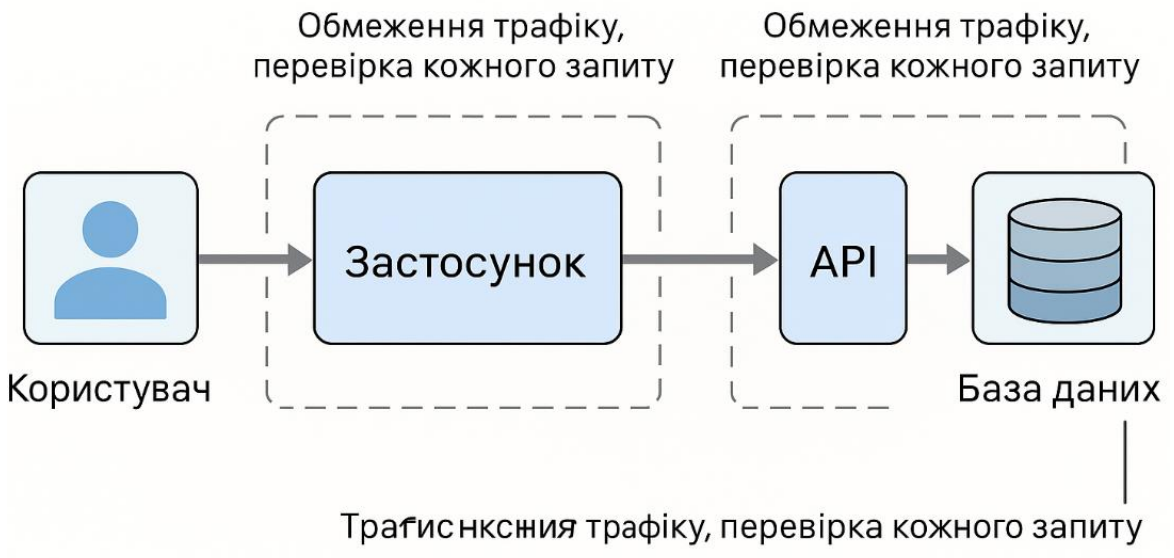

Рисунок 1.1 – Схема Zero Trust або мікросегментації

Принцип відокремленості процесів

Принцип відокремленості процесів є одним із фундаментальних підходів у забезпеченні архітектурної безпеки великих програмних систем. Його сутність полягає в тому, що кожен окремий процес, служба або компонент системи повинен функціонувати в ізольованому середовищі, що виключає або мінімізує можливість впливу одного процесу на інші. Такий підхід дозволяє локалізувати загрози та обмежити масштаби потенційного

ураження у разі компрометації одного з компонентів. У контексті великих розподілених або багаторівневих систем це має особливо важливе значення, оскільки складність взаємодії між численними модулями часто є джерелом критичних уразливостей.

Ізоляція процесів може бути реалізована на різних рівнях: апаратному, операційної системи, мережевому та прикладному. На апаратному рівні використовуються віртуальні машини або контейнери, які створюють логічно відокремлені середовища, що дозволяють запускати процеси незалежно один від одного. Контейнеризація за допомогою технологій на кшталт Docker, Kubernetes або Podman є однією з найефективніших форм практичної реалізації принципу відокремленості. Вона дозволяє забезпечити швидке масштабування, централізоване управління та одночасно гарантувати, що злам або відмова одного контейнера не вплине на решту системи.

Операційна система, у свою чергу, забезпечує розмежування доступу до ресурсів через механізми контролю доступу до оперативної пам’яті, файлової системи, мережевих сокетів, обчислювальних ядер тощо. Наприклад, у Linux використовуються такі модулі, як SELinux або AppArmor, які дозволяють чітко визначати політики доступу між процесами. Така сегментація дозволяє зменшити ризик того, що зловмисний код, запущений у рамках одного процесу, отримає доступ до чутливих даних іншого. Також важливо відзначити роль систем розмежування на рівні ідентифікаторів користувачів (UID, GID), які застосовуються до кожного окремого процесу, що дозволяє точно регулювати повноваження на доступ до системних ресурсів [36].

На прикладному рівні відокремленість досягається через поділ логіки додатків на незалежні компоненти, що взаємодіють між собою виключно через стандартизовані API або черги повідомлень. Наприклад, у мікросервісній архітектурі кожен сервіс реалізує чітко визначену функцію та взаємодіє з іншими через HTTP, gRPC або брокери повідомлень (Kafka, RabbitMQ). Цей підхід дозволяє не лише масштабувати систему, але й обмежити наслідки інцидентів безпеки, оскільки порушення в одному сервісі не впливають

безпосередньо на інші. Зокрема, кожен мікросервіс може мати свій набір облікових даних, свій окремий простір зберігання, окремий журнал аудиту та окремі механізми авторизації.

Велике значення у реалізації принципу відокремленості має також застосування концепції «least communication» – тобто обмеження комунікації між процесами лише до тих даних, які є необхідними. Це означає, що не повинно бути «загальних областей пам’яті», «загальних конфігурацій» або «загальних файлів», до яких можуть звертатися кілька процесів одночасно без належного контролю. Замість цього доцільним є використання шифрування, автентифікації, тимчасових токенів або одноразових ключів для захищеної комунікації між сервісами.

Ще одним аспектом є мережеве ізолювання, яке забезпечує логічне розділення трафіку на рівні внутрішньої інфраструктури. Це може бути реалізовано за допомогою віртуальних мереж, VLAN, приватних підмереж або міжмережевих екранів. Наприклад, фронтенд–сервіси можуть мати доступ лише до API–шлюзів, але не до бекенд–баз даних, тоді як бази даних можуть взаємодіяти лише з обмеженим колом IP–адрес. Мережевий контроль доступу дозволяє зменшити ризик горизонтального переміщення атакуючого після початкового проникнення у систему.

Таким чином, принцип відокремленості процесів є не лише методологічним підходом, а й сукупністю технічних і адміністративних заходів, спрямованих на побудову модульної, керованої, безпечної системи. Його реалізація у великих програмних системах дозволяє не лише покращити рівень кіберстійкості, але й спрощує моніторинг, оновлення та аудит компонентів. У сучасних умовах, коли атаки стають дедалі складнішими та багатоетапними, здатність ізолювати наслідки інциденту до окремого процесу або контейнера може мати вирішальне значення для стабільності та надійності всієї програмної інфраструктури [24].

Принцип моніторингу та аудиту

У сучасних архітектурах безпеки великих програмних систем принцип моніторингу та аудиту є одним з основоположних і водночас найбільш динамічних у плані реалізації. Цей принцип передбачає постійний збір, збереження та аналіз інформації про події, що відбуваються в системі, з метою своєчасного виявлення порушень політик безпеки, індикаторів атак, спроб несанкціонованого доступу або внутрішніх аномалій. У масштабних програмних рішеннях, які працюють у режимі 24/7, а також обробляють критичні дані, безперервний моніторинг і аудит є не розкішшю, а обов'язковою умовою відповідності міжнародним стандартам, таким як ISO/IEC 27001, NIST SP 800–53, SOC 2, GDPR та іншим.

Моніторинг безпеки охоплює широкий спектр інструментів та практик. Насамперед, це системи збору логів (журналів подій), які фіксують усі ключові дії користувачів, сервісів і компонентів системи. До таких дій відносяться спроби входу, зміни конфігурацій, доступ до конфіденційної інформації, підозрілі запити до API, нестандартне використання ресурсів тощо. Зазвичай ці логи зберігаються централізовано та поділяються на категорії: лог подій користувача, системний лог, лог помилок, мережевий трафік, лог аутентифікації тощо. Важливо забезпечити їх цілісність, недоступність для змін користувачами, що не мають спеціальних прав, а також налаштувати автоматичне ротаційне збереження та архівування.

Розширеною формою реалізації цього принципу є впровадження SIEM–систем (Security Information and Event Management), які автоматизують процес збору, кореляції, аналізу та реагування на події безпеки. Такі системи використовують алгоритми обробки великих даних, машинне навчання та евристичні правила для виявлення складних інцидентів, які неможливо побачити в межах одного журналу. Наприклад, якщо відбулося п'ять спроб входу в обліковий запис адміністратора з різних IP-адрес протягом хвилини –

це може бути індикатором брутфорс-атаки. SIEM-система виявить це як підозрілу активність і повідомить службу безпеки [19].

Аудит, на відміну від моніторингу, спрямований на ретроспективний аналіз діяльності користувачів і системних процесів. Він дозволяє виявляти відхилення від політик, інциденти безпеки, перевіряти відповідність стандартам, готувати звіти для регуляторів або внутрішнього контролю. У контексті великих програмних систем особливо важливим є аудит змін до системних налаштувань, прав доступу, облікових записів і застосунків. Такий аудит повинен бути регулярним, документованим та незалежним від безпосередніх виконавців. Крім того, має бути можливість відновити повну історію взаємодії користувача з системою навіть через тривалий період часу.

Сучасні інструменти забезпечують також інтеграцію моніторингу з системами реагування на інциденти (SOAR – Security Orchestration, Automation and Response), що дозволяє не лише виявляти загрози, але й автоматизувати певні дії у відповідь – наприклад, блокування IP-адреси, вимкнення користувача, створення задачі для інженера безпеки тощо. Це знижує час реагування, мінімізує вплив атак і дає можливість командам зосередитися на складніших аспектах аналітики.

Велике значення має також візуалізація даних моніторингу. Dashboards, графіки, сповіщення в реальному часі та heatmaps дозволяють адміністраторам систем безпеки швидко ідентифікувати аномалії. Інструменти на кшталт Grafana, Kibana, Splunk, Zabbix використовуються для побудови кастомізованих панелей моніторингу, що охоплюють критичні ділянки системи: мережеву активність, навантаження, частоту помилок, активність користувачів, інтеграційні потоки тощо.

Отже, принцип моніторингу та аудиту є ключовим у побудові архітектури безпеки великих програмних систем. Його реалізація дозволяє не лише виявляти й аналізувати інциденти, але й формувати культуру прозорості, відповідальності та технічного контролю в межах усієї організації. Залежність сучасних IT-систем від складних і динамічних середовищ зумовлює

необхідність у постійному оновленні інструментарію моніторингу, удосконаленні процедур аудиту та впровадженні практик швидкого реагування. Систематичний підхід до моніторингу та аудиту забезпечує своєчасне виявлення загроз і знижує загальні ризики, пов'язані з втратою даних, компрометацією систем або порушенням конфіденційності [10].

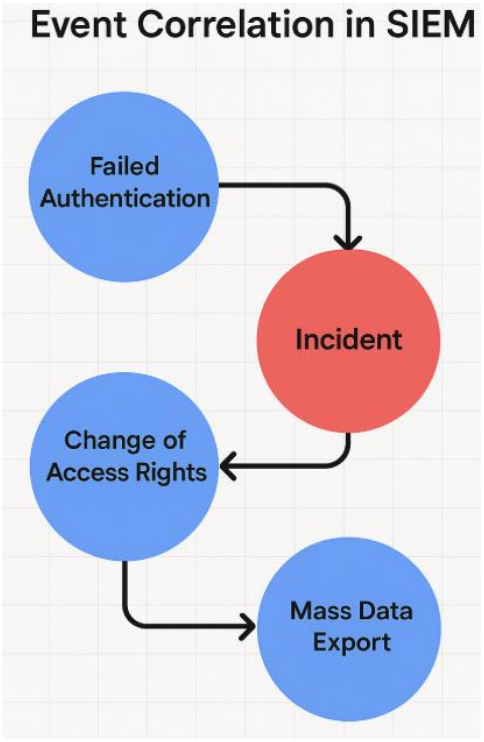

Рисунок 1.2 – Приклад кореляції подій у SIEM

Принцип захисту даних

Принцип захисту даних є ключовим елементом побудови архітектури безпеки великих програмних систем, оскільки саме дані – структуровані й неструктуровані, конфіденційні чи публічні – є основною цінністю будь–якої сучасної інформаційної інфраструктури. У масштабних програмних рішеннях обсяги оброблюваних даних можуть сягати терабайтів або навіть петабайтів, а сама інформація циркулює між численними модулями, користувачами, хмарними сервісами, локальними сховищами та API. У таких умовах забезпечення цілісності, доступності та конфіденційності даних стає

першочерговим завданням, від якого залежить не лише стабільність роботи системи, а й репутація організації, що її експлуатує, а також дотримання законодавчих вимог, таких як GDPR, CCPA, HIPAA, ISO/IEC 27001, український Закон «Про захист персональних даних» та інші.

Принцип захисту даних охоплює комплекс технічних, організаційних та адміністративних заходів, спрямованих на недопущення несанкціонованого доступу, витоку, зміни або знищення даних. На технічному рівні першочерговим заходом є шифрування, яке має застосовуватися як до даних, що зберігаються (at rest), так і до даних, що передаються (in transit). Для захисту даних у стані зберігання використовуються алгоритми симетричного шифрування, наприклад AES-256, із захищеним управлінням ключами через спеціалізовані модулі HSM (Hardware Security Modules). Передача даних повинна здійснюватися виключно через захищені протоколи (TLS 1.2/1.3, HTTPS, SFTP), а використання застарілих стандартів (наприклад, SSL, FTP, MD5) має бути повністю виключене.

Особливої уваги потребує захист персональних даних (PII), фінансової інформації, медичних записів, конфіденційних комерційних документів, внутрішніх службових листувань та іншої інформації, що може бути предметом регуляторного контролю або має високу бізнес–цінність. Для таких даних застосовуються додаткові заходи, такі як токенізація, псевдонімізація, маскування даних у середовищі розробки, багаторівневі системи контролю доступу на основі ролей та атрибутів. Наприклад, дані клієнтів у базі можуть бути представлені у вигляді одноразових токенів для аналітичних сервісів, а лише сертифіковані працівники із дозволом на обробку персональних даних можуть бачити їх у незашифрованому вигляді [33].

У великих програмних системах, де численні сервіси взаємодіють через API, важливим є захист інтерфейсів передачі даних. Це досягається впровадженням механізмів авторизації (OAuth 2.0, OpenID Connect), обмеженням частоти запитів (rate limiting), валідацією вхідних параметрів, впровадженням фільтрів контенту, застосуванням протоколів захисту від XSS,

SQL injection, CSRF та інших атак. Також доцільним є запровадження систем DLP (Data Loss Prevention), які дозволяють виявляти підозрілі дії, пов’язані з масовим копіюванням, надсиланням чи редагуванням конфіденційної інформації.

Не менш критичним є забезпечення резервного копіювання та можливості відновлення даних (backup & disaster recovery). Архітектура безпеки повинна передбачати наявність регулярних резервних копій, що зберігаються в географічно рознесених зонах, мають шифрування та не можуть бути змінені навіть адміністраторами системи без проходження багатофакторної автентифікації. Важливим є також періодичне тестування сценаріїв відновлення даних після збоїв або атак (наприклад, програм-вимагачів), оскільки лише наявність копії не гарантує її працездатність.

З точки зору організаційного підходу принцип захисту даних передбачає формування політик класифікації даних, згідно з якими визначається рівень конфіденційності кожного типу інформації. Наприклад, документи можуть поділятися на загальнодоступні, внутрішні, конфіденційні, критичні. Для кожного рівня визначаються вимоги щодо зберігання, доступу, шифрування, передачі, утилізації. Користувачі системи повинні бути поінформовані про ці політики, пройти відповідне навчання та нести відповідальність за порушення норм збереження інформації.

Особливу увагу слід приділяти питанням логування та моніторингу доступу до даних. Усі спроби перегляду, редагування або передачі важливих даних повинні реєструватися із зазначенням часу, ідентифікатора користувача, джерела доступу та типу операції. Централізовані системи моніторингу повинні виявляти аномальні патерни поведінки, наприклад масові запити до однієї таблиці, надмірну активність у неробочі години або звернення до заборонених файлів. У таких випадках система повинна негайно генерувати сповіщення безпеки, блокувати запит або переводити обліковий запис у тимчасово заблокований стан.

Необхідно також забезпечити контроль за життєвим циклом даних – від моменту створення до моменту їхнього знищення. Це включає контроль за термінами зберігання відповідно до вимог регуляторів (наприклад, термін зберігання податкової звітності або медичних карток), процедури безпечної утилізації (наприклад, перезапис даних кілька разів згідно з алгоритмами DoD або Gutmann) та облік усіх дій, пов'язаних з видаленням інформації. Дані не повинні залишатися на пристроях після завершення їх використання, особливо якщо йдеться про зовнішні накопичувачі, мобільні пристрої або хмарні сервіси.

Дотримання принципу захисту даних також вимагає регулярного проведення аудитів безпеки, перевірок на проникнення (penetration testing), аналізу коду (static code analysis) та тестування конфігурацій. Усі результати мають документуватись, а виявлені недоліки – усуватись у найкоротші терміни з подальшою перевіркою результату. Важливим є впровадження підходу security by design – тобто врахування вимог до безпеки даних ще на етапі проєктування архітектури програмної системи.

Таким чином, принцип захисту даних – це багатовимірна концепція, яка вимагає комплексного підходу з боку архітекторів, розробників, адміністраторів, юристів та фахівців з кібербезпеки. В умовах постійного зростання обсягів оброблюваної інформації, активізації кібератак, підвищених вимог регуляторів та зростання кількості партнерських інтеграцій, захист даних є не лише технологічним викликом, а й стратегічним пріоритетом для організацій, які прагнуть забезпечити довіру користувачів, конкурентоспроможність і відповідність сучасним нормам інформаційної безпеки [38].

Принцип реагування на інциденти

Принцип реагування на інциденти є невід'ємною складовою архітектури безпеки великих програмних систем, оскільки навіть найкращі превентивні

заходи не можуть повністю гарантувати захист від усіх можливих загроз. У сучасному кіберсередовищі атаки постійно еволюціонують, використовуючи нові уразливості, соціальну інженерію, помилки конфігурацій та людський фактор. Саме тому важливо не лише запобігати порушенням безпеки, а й бути готовими до їх швидкого виявлення, аналізу, локалізації та усунення наслідків. Реагування на інциденти забезпечує оперативне повернення системи до стабільного стану, мінімізацію втрат і захист репутації організації.

У великих програмних системах, які обслуговують тисячі або мільйони користувачів і мають розподілену архітектуру, принцип реагування на інциденти охоплює організацію комплексного процесу, що включає визначення потенційних сценаріїв загроз, створення планів дій, виділення відповідальних осіб, автоматизацію процедур та регулярне тестування готовності. Насамперед має бути сформована команда реагування на інциденти – CSIRT (Computer Security Incident Response Team), яка відповідає за управління всіма етапами роботи з інцидентами: від моменту їх виявлення до остаточного усунення та формування підсумкових звітів.

Процес реагування на інциденти умовно поділяється на кілька фаз. Перша – це виявлення інциденту, яке може здійснюватися автоматичними засобами моніторингу, користувачами, адміністраторами або зовнішніми партнерами. Для цього використовуються SIEM–системи, аналіз логів, поведінкові алгоритми, виявлення аномалій, повідомлення від користувачів, результати сканування вразливостей тощо. Важливо, щоб система виявлення працювала в режимі реального часу та підтримувала кореляцію подій із різних джерел. Наприклад, несанкціонований доступ до адміністративного інтерфейсу з IP-адреси, яка раніше не використовувалася, може бути автоматично розцінений як інцидент і переданий на обробку [27].

Другою фазою є класифікація інциденту, тобто визначення його рівня критичності, масштабу, джерела та потенційного впливу. Це дозволяє визначити пріоритети реагування. Наприклад, фішингова атака, яка призвела до компрометації одного облікового запису, має інший рівень критичності, ніж

витік бази даних клієнтів або злам внутрішньої ERP-системи. Класифікація допомагає сформувати ефективний план дій, визначити необхідні ресурси, залучити відповідних фахівців і обрати відповідні технічні інструменти.

Третя фаза – стримування інциденту. На цьому етапі основним завданням є запобігання подальшому поширенню впливу або шкоди. Це може включати тимчасове блокування облікових записів, відключення сегментів мережі, припинення дії API, ізоляцію зламаних серверів або контейнерів, відключення підозрілих скриптів. Швидке та ефективне стримування дозволяє виграти час для детального аналізу та уникнути каскадних наслідків. У великих системах використовуються автоматизовані правила реагування, які дозволяють миттєво реагувати на типові інциденти – наприклад, при виявленні спроби SQL-ін’єкції система блокує IP-адресу атакуючого і створює інцидент у системі управління подіями.

Четверта фаза – усунення наслідків, тобто повне або часткове відновлення роботи системи до нормального стану. Цей етап включає оновлення конфігурацій, виправлення вразливостей, відновлення даних із резервних копій, перевстановлення компонентів, які були скомпрометовані, та інші дії, спрямовані на відновлення цілісності системи. У деяких випадках цей етап вимагає взаємодії з розробниками, юридичними відділами, зовнішніми експертами. У разі виявлення шкідливого коду важливо не лише видалити його, а й зрозуміти механізм проникнення, щоб унеможливити повторення.

П’ята фаза – відновлення та повернення до стандартного режиму. Вона включає перевірку повної стабільності системи, поступове повернення до штатного функціонування, повторну перевірку безпеки, відкриття сервісів, інформування користувачів. Якщо інцидент був публічним або зачіпав персональні дані, необхідно дотриматися вимог звітності перед державними регуляторами або постраждалими користувачами, що є обов’язковим за законодавством багатьох країн.

Завершальною, але не менш важливою фазою є документування та аналіз інциденту. Команда реагування повинна сформувати звіт, у якому буде

зазначено причини інциденту, дії, що були вжиті, хронологію подій, оцінку шкоди, перелік зацікавлених сторін, рекомендації щодо уникнення подібних ситуацій у майбутньому. Цей звіт слугує основою для оновлення політик безпеки, налаштувань моніторингу, процедур реагування, навчання персоналу.

Принцип реагування на інциденти також передбачає проведення навчань, моделювання ситуацій (tabletop exercises), тестування сценаріїв (red team / blue team), автоматизацію дій за допомогою платформ типу SOAR. Усе це дозволяє підвищити готовність до реальних загроз, зменшити час реагування, забезпечити координацію між технічними та адміністративними підрозділами. У великих системах, які обслуговують критичну інфраструктуру, окремі вимоги до реагування можуть бути закріплені в нормативно–правових актах або в контрактах із партнерами (наприклад, SLA – service level agreement).

У важливу категорію входять також інциденти, спричинені внутрішніми загрозами – помилки персоналу, зловмисні дії співробітників, порушення процедур. У таких випадках особливої уваги набуває аспект цифрових доказів, які можуть бути використані у дисциплінарних або судових розглядах. Саме тому всі дії під час реагування мають бути задокументовані, часово синхронізовані, підписані електронними підписами або збережені в незмінному вигляді у SIEM [36].

РОЗДІЛ 2
ОСНОВНІ МЕХАНІЗМИ ЗАХИСТУ ВЕЛИКИХ ПРОГРАМНИХ СИСТЕМ

Аутентифікація та авторизація

Аутентифікація та авторизація є двома ключовими механізмами, які забезпечують контроль доступу до ресурсів великих програмних систем. Вони є основою концепції інформаційної безпеки, де критичне значення має перевірка особи (аутентифікація) та визначення її прав (авторизація). Без чіткої реалізації цих процесів неможливо гарантувати цілісність, конфіденційність і доступність інформаційних активів. У великих програмних системах ці механізми повинні бути масштабованими, гнучкими, сумісними з різними джерелами ідентифікації та підтримувати багатоетапну обробку даних про користувача.

Аутентифікація – це процес перевірки особи користувача або пристрою, який намагається отримати доступ до системи. Основні методи аутентифікації класифікуються за категоріями: «те, що ви знаєте» (паролі, PIN-коди), «те, що ви маєте» (токени, смарт-карти), «те, ким ви є» (біометрія). У сучасних системах широко впроваджується багатофакторна аутентифікація (MFA), яка комбінує щонайменше два з наведених чинників. Наприклад, користувач вводить пароль (знання) та підтверджує вхід через мобільний додаток (володіння). Це значно ускладнює несанкціонований доступ, особливо у разі компрометації одного з факторів.

Авторизація, у свою чергу, визначає, що саме дозволено робити автентифікованому користувачеві в системі. Це можуть бути операції з читанням, записом, видаленням, адмініструванням тощо. Реалізація авторизації здійснюється через моделі управління доступом, серед яких найпоширенішими є RBAC (Role-Based Access Control), ABAC (Attribute-Based Access Control) та PBAC (Policy-Based Access Control). У великих

програмних системах особливо актуальним є динамічне керування доступом із урахуванням контексту – наприклад, час входу, тип пристрою, геолокація, історія взаємодії [31].

Нижче наведено порівняльну таблицю 2.1 основних механізмів аутентифікації.

Таблиця 2.1 – Основні методи аутентифікації у великих програмних системах

№	Метод аутентифікації	Опис	Рівень безпеки	Типові приклади використання
1	Пароль	Статичне значення, яке знає користувач	Низький	Особистий кабінет, внутрішні сервіси
2	Одноразовий код (OTP)	Код, що генерується динамічно, діє протягом обмеженого часу	Середній	Вхід до банківського додатку
3	Біометрична аутентифікація	Верифікація за відбитком пальця, сітківкою, обличчям тощо	Високий	Смартфони, прикордонні системи
4	Смарт-карта/токен	Фізичний пристрій або USB-ключ, що містить сертифікат або код доступу	Високий	Державні та військові системи
5	Багатофакторна аутентифікація	Комбінація двох або більше факторів	Дуже високий	Корпоративні середовища, хмара, VPN

Прикладом практичного впровадження може слугувати корпоративна платформа управління документами, де працівники мають доступ до різних модулів системи – фінансового, юридичного, аналітичного. Кожен з цих модулів містить чутливу інформацію, доступ до якої слід регулювати відповідно до посадових обов’язків. Для цього впроваджується система RBAC: аналітики мають лише читання звітів, юристи – редагування контрактів, а адміністратори – повний доступ. Усі дії користувачів фіксуються, а доступ до системи обов’язково здійснюється через MFA з підтвердженням через корпоративний мобільний застосунок.

Нижче подано приклад таблиці 2.2 ролей і прав доступу:

Таблиця 2.2 – Матриця ролей і доступів у системі електронного документообігу

№	Роль користувача	Модуль "Фінанси"	Модуль "Юридичний"	Модуль "Звіти"	Доступ до системних налаштувань
1	Адміністратор	Читання/Запис	Читання/Запис	Повний доступ	Повний доступ
2	Аналітик	Читання	Немає доступу	Повний доступ	Немає доступу
3	Юрист	Немає доступу	Читання/Запис	Читання	Немає доступу
4	Гість	Обмежене читання	Обмежене читання	Обмежене	Немає доступу

У складних хмарних системах додатково впроваджується federated identity – тобто можливість використання сторонніх постачальників ідентичності (наприклад, Google, Microsoft Azure, Okta), що дозволяє централізовано керувати обліковими записами користувачів і знижує ризики, пов’язані з дублюванням даних або помилковим призначенням прав.

Іншим критичним компонентом є SSO (Single Sign-On) – механізм, який дозволяє користувачеві один раз пройти аутентифікацію і отримати доступ до багатьох пов’язаних ресурсів без повторного введення облікових даних. Це підвищує зручність, зменшує кількість помилок введення паролів і дозволяє краще управляти сесіями доступу [29].

З технічної точки зору, для авторизації у розподілених системах часто використовуються токени доступу – зокрема, формати JWT (JSON Web Token), які містять зашифровану інформацію про користувача, термін дії токена, роль, дозволені дії. Системи мають валідувати токени на кожному запиті та перевіряти їхню актуальність і відповідність політикам доступу.

У сучасному DevSecOps-підході важливо також враховувати автоматизовану аутентифікацію між мікросервісами. Наприклад, для взаємодії між сервісами A і B в архітектурі Kubernetes застосовується mutual TLS, при якому обидві сторони мають сертифікати і верифікують одна одну перед обміном даними. Це забезпечує не лише ідентифікацію, а й захист каналів зв’язку від перехоплення або підміни.

Ще одним викликом є управління привілеями користувачів протягом їхнього життєвого циклу в системі. Нові користувачі мають отримувати доступ лише після проходження регламентованої процедури погодження. При зміні посади або проекту – їхні права мають бути автоматично скориговані, а після звільнення – повністю відкликані з перевіркою видалення облікових даних з усіх пов’язаних систем.

Таким чином, аутентифікація та авторизація у великих програмних системах є складним, багатокомпонентним процесом, що вимагає інтеграції численних технологій, підтримки стандартів безпеки, гнучкого управління, журналювання подій доступу, а також автоматизації життєвого циклу ідентифікаційної інформації. Їх ефективна реалізація зменшує площу потенційної атаки, запобігає помилковому призначенню прав і забезпечує контрольований, прогнозований доступ до ресурсів системи. Практичне впровадження цих принципів має бути однією з пріоритетних задач для фахівців з архітектури IT-безпеки в умовах швидкої цифрової трансформації та зростання кіберзагроз [4].

2.2 Шифрування та моніторинг

Шифрування та моніторинг є ключовими інструментами забезпечення інформаційної безпеки у великих програмних системах, що функціонують у складному, динамічному та часто відкритому інформаційному середовищі. Шифрування виконує завдання захисту даних як у стані зберігання, так і під час передачі, тоді як моніторинг дозволяє забезпечити постійний контроль за діями в системі, виявлення аномалій, інцидентів і потенційних атак у реальному часі. Обидва ці процеси є невіддільними компонентами комплексного підходу до безпеки, який враховує як технічні, так і організаційні аспекти захисту критичних активів.

Шифрування даних – це процес перетворення інформації в зашифровану форму, яку не можна прочитати без відповідного ключа розшифрування. У

великих програмних системах шифрування використовується на всіх рівнях – від зберігання файлів і баз даних до захисту трафіку між мікросервісами або компонентами, що працюють у різних дата–центрах або хмарних середовищах. Найпоширенішим стандартом є AES (Advanced Encryption Standard), зокрема з довжиною ключа 256 біт, який забезпечує високий рівень криптографічної стійкості. У випадках, коли необхідна взаємна аутентифікація або захист підпису, використовуються асиметричні алгоритми – наприклад RSA або ECC.

Загальна практика захисту даних включає шифрування у стані спокою (data at rest), під час передачі (data in transit) і в деяких випадках – у стані обробки (data in use). Шифрування даних у стані спокою зазвичай реалізується на рівні диска або файлової системи, наприклад за допомогою BitLocker, LUKS, EFS або вбудованих механізмів шифрування баз даних (наприклад, TDE – Transparent Data Encryption). Для захисту даних у стані передачі застосовуються такі протоколи, як TLS 1.2/1.3, SSH, HTTPS, SFTP, IPsec. Контроль за їх використанням повинен бути реалізований у політиках безпеки та перевірятись системами моніторингу [15].

Нижче наведено таблицю 2.3 з класифікацією методів шифрування та сфер їх застосування.

Таблиця 2.3 – Основні типи шифрування у великих програмних системах

№	Тип шифрування	Алгоритми	Сфера застосування	Особливості реалізації
1	Симетричне	AES, Blowfish, ChaCha20	Захист файлів, БД, трафіку в реальному часі	Висока швидкість, складність управління ключами
2	Асиметричне	RSA, ECC, ElGamal	Захист каналів зв’язку, цифровий підпис	Висока обчислювальна складність, довгі ключі
3	Гібридне	TLS (AES + RSA), PGP	Веб–трафік, електронна пошта	Комбінує швидкість і безпеку
4	Шифрування на рівні БД	TDE, Always Encrypted	SQL Server, Oracle, PostgreSQL	Прозоре для користувача, вимагає конфігурації
5	Поле–в–шифрування	Column–Level Encryption	Селективний захист полів у таблицях БД	Дозволяє гнучко керувати конфіденційністю

Реальна ситуація, яка демонструє значення шифрування – це хмарна CRM-система, що обробляє персональні дані клієнтів. Усі дані зберігаються у PostgreSQL з активованим шифруванням TDE. Доступ до бази можливий лише з IP–адрес внутрішньої мережі через TLS-захищене з'єднання. Кожен клієнтський запис додатково шифрується симетричним ключем, який зберігається в HSM. Це унеможливлює читання інформації навіть у випадку витоку бази. Ключі змінюються кожні 30 днів, журнал обігу ключів ведеться автоматично і перевіряється службою безпеки.

Що стосується моніторингу – це постійний процес спостереження за подіями в системі, який дозволяє виявити порушення політик безпеки, збої, аномалії у поведінці користувачів або системних компонентів. У масштабних програмних системах обсяг подій може становити мільйони рядків логів на день, тому без застосування автоматизованих рішень цей процес стає непридатним для оперативного реагування. Саме тому впроваджуються SIEM-системи (Security Information and Event Management), які збирають логи з усіх компонентів, аналізують їх за заданими правилами та сигналізують про інциденти.

Моніторинг може охоплювати кілька рівнів: мережевий (спроби сканування портів, підозрілі IP), системний (невдалі спроби входу, запуск підозрілих процесів), прикладний (аномальна кількість запитів до API, незвична поведінка користувача), баз даних (масове читання таблиць, підозрілі запити), доступу до файлів (видалення великого обсягу даних, копіювання конфіденційних документів).

Реалізація практичної частини вимагає впровадження логування, налаштування збору подій та створення системи попереджень. Наприклад, при розробці системи документообігу, де працівники мають доступ до фінансової звітності, впроваджується механізм логування всіх спроб експорту файлів PDF, які містять критичні дані. Якщо один користувач експортує понад п'ять звітів за 10 хвилин – генерується тривожне повідомлення у SIEM. Аналогічно, якщо система бачить кілька невдалих спроб входу з IP-адрес за межами

України – автоматично блокується обліковий запис, надсилається повідомлення на пошту служби безпеки, а також зберігається знімок сесії для подальшого аналізу.

Нижче в табл. 2.4 представлена типова структура моніторингу у великій системі.

Таблиця 2.4 – Рівні моніторингу та їх призначення

№	Рівень моніторингу	Основні метрики/події	Інструменти спостереження
1	Мережевий	Трафік, підключення, спроби сканування	Suricata, Zeek, Wireshark
2	Системний	CPU, пам’ять, дискові I/O, процеси, логіни	Prometheus, Zabbix, Grafana
3	Прикладний	Кількість запитів, API, відхилення у поведінці	ELK Stack, Splunk
4	Баз даних	Аудит запитів, час виконання, великі SELECT/DELETE	pgAudit, Oracle Audit, Query Monitor
5	Користувацький рівень	Дії користувача, невдалі входи, спроби ескалації	SIEM, UEBA (User & Entity Behavior Analytics)

У сучасних системах часто використовують також моніторинг цілісності файлів (FIM – File Integrity Monitoring). Такий підхід дозволяє виявляти зміни в конфігураційних файлах, скриптах, бібліотеках, що може свідчити про атаку або несанкціоноване втручання. Наприклад, система регулярно перевіряє контрольні суми критичних файлів – якщо checksum змінено без відповідного запису в логах змін, інцидент фіксується автоматично.

У підсумку, шифрування забезпечує базовий рівень конфіденційності та захисту даних, тоді як моніторинг дозволяє реагувати на спроби порушення цих гарантій. Ці два елементи є взаємопов’язаними: шифрування без моніторингу не дозволяє виявити спроби доступу або розшифрування, тоді як моніторинг без шифрування залишає дані відкритими до компрометації. Впровадження надійних систем шифрування та ефективного моніторингу – це необхідна умова для функціонування безпечного програмного середовища, здатного витримати сучасні виклики кібербезпеки [17].

РОЗДІЛ 3
АНАЛІЗ ПОТЕНЦІЙНИХ ЗАГРОЗ ТА ВРАЗЛИВОСТЕЙ, ЯКІ МОЖУТЬ ВПЛИНУТИ НА БЕЗПЕКУ ВЕЛИКИХ ПРОГРАМНИХ СИСТЕМ

Використання вразливостей програмного забезпечення для здійснення кібератак

Використання вразливостей програмного забезпечення для здійснення кібератак є одним із найпоширеніших і найнебезпечніших методів порушення безпеки великих програмних систем. Уразливості – це дефекти в коді, архітектурі, конфігурації або логіці програмного забезпечення, які можуть бути використані зловмисниками з метою несанкціонованого доступу, підвищення привілеїв, викрадення або знищення даних, встановлення шкідливого коду чи порушення працездатності системи. У великих програмних системах, які мають складну багаторівневу структуру, містять тисячі модулів і обробляють великі обсяги інформації, наявність хоча б однієї критичної вразливості може призвести до катастрофічних наслідків.

Уразливості можуть виникати як у власному коді організації, так і у сторонньому програмному забезпеченні, бібліотеках з відкритим кодом, фреймворках, компонентах операційної системи або середовища виконання. Зловмисники активно використовують публічні бази даних, такі як CVE (Common Vulnerabilities and Exposures) або NVD (National Vulnerability Database), де регулярно публікуються звіти про нові знайдені вразливості з детальним описом умов їх експлуатації. Багато атак відбувається відразу після публікації нових CVE, що змушує компанії оперативно реагувати шляхом оновлення компонентів, виправлення конфігурацій або впровадження тимчасових обмежень.

Найбільш типові класи вразливостей включають буферні переповнення (buffer overflow), ін’єкції SQL, XSS (cross-site scripting), CSRF (cross-site request forgery), відсутність валідації вхідних даних, неправильне керування

сесіями, вразливості пов’язані з криптографією (наприклад, використання застарілих або слабких алгоритмів), проблеми з правами доступу, а також логічні помилки в бізнес-процесах. Більшість таких уразливостей виникає через нехтування принципами безпечного програмування або недосконаліть механізмів тестування під час життєвого циклу розробки ПЗ [8].

Нижче наведено типову класифікацію вразливостей і способів їх використання.

Таблиця 3.1 – Класи вразливостей програмного забезпечення та можливі наслідки їх експлуатації

№	Тип вразливості	Опис	Можливі наслідки	Типові приклади інструментів атак
1	SQL–ін’єкція	Можливість вставки шкідливих SQL–запитів	Викрадення або знищення даних, ескалація прав	sqlmap, Havij
2	XSS	Вставка шкідливого JS у вебсторінку	Крадіжка cookies, перенаправлення користувача	BeEF, XSStrike
3	Переповнення буфера	Запис за межі виділеної пам’яті	Віддалене виконання коду, аварійне завершення	Metasploit, Immunity Debugger
4	Відсутність контролю доступу	Недостатня перевірка прав користувача	Несанкціонований доступ до даних або функцій	Burp Suite, ZAP Proxy
5	Використання застарілих бібліотек	Наявність уразливих версій компонентів	Повне компрометування системи через відомі CVE	Nmap, Nessus, OpenVAS
6	CSRF	Підміна запиту з боку користувача	Несанкціоноване виконання дій	OWASP ZAP, CSRFTester
7	Вразливості серіалізації	Маніпуляція з об’єктами під час передачі	Виконання довільного коду	ysoserial, Java Deserialization Scanner

Впровадження захисту від вразливостей передбачає також створення процедур регулярного оновлення програмного забезпечення, автоматизацію сканування коду (SAST, DAST), обмеження прав доступу до критичних компонентів, розділення середовищ (dev/test/prod), валідацію вхідних даних та аудит усіх викликів API. Одним із ефективних підходів є використання WAF

(Web Application Firewall), який дозволяє фільтрувати шкідливі запити до веб-серверів і попереджати реалізацію відомих шаблонів атак.

Отже, використання вразливостей програмного забезпечення залишається одним з найактуальніших векторів кібератак на великі програмні системи. Для ефективного захисту необхідно впроваджувати багаторівневий підхід, що включає безпечну розробку, регулярне оновлення компонентів, використання автоматизованих засобів виявлення уразливостей, а також постійне підвищення обізнаності розробників і DevOps-команд щодо сучасних методів експлуатації вразливостей. Тільки завдяки системному та превентивному підходу можна знизити ризик успішного використання відомих і потенційних уразливостей, захистивши програмну інфраструктуру від руйнівних наслідків кібератак [39].

Огляд шкідливих програм та вірусів, що можуть спричинити втрату даних або завдати шкоду програмній системі

Шкідливе програмне забезпечення (malware) є одним із найнебезпечніших інструментів у руках кіберзлочинців, що використовується для порушення конфіденційності, цілісності та доступності інформації у великих програмних системах. З ускладненням архітектури IT–інфраструктур, активним переходом до хмарних платформ і мікросервісної архітектури, а також через залежність від сторонніх бібліотек, систем CI/CD і мобільних клієнтів, загроза зараження шкідливим ПЗ стала ще більш актуальною. Malware здатне інфільтрувати систему, залишаючись непоміченим протягом тривалого часу, виконувати фонові дії, красти дані, модифікувати конфігурації, викликати відмову в обслуговуванні або навіть повністю виводити програмну систему з ладу.

Шкідливі програми поділяються на багато категорій залежно від механізму дії, цілей атакуючих і технологій, які вони використовують. До основних типів malware відносяться віруси, трояни, черв’яки, програми–

вимагачі (ransomware), бекдори (backdoors), шпигунське ПЗ (spyware), кейлогери, рекламне ПЗ (adware), руткіти та ботнет-клієнти. У деяких випадках шкідливі програми комбінують функції кількох типів, утворюючи так звані поліморфні загрози, які змінюють свою структуру для уникнення виявлення антивірусами [34].

Нижче наведено типову класифікацію шкідливих програм за типами та механізмами дії.

Таблиця 3.2 – Класифікація шкідливого ПЗ за функціональністю

№	Тип шкідливого ПЗ	Опис дії	Потенційна шкода	Типові приклади
1	Вірус	Приєднується до файлів, поширюється при їх виконанні	Пошкодження файлів, порушення логіки додатків	CIH, FileInfector
2	Черв’як	Поширюється самостійно через мережу або знімні носії	Швидке зараження систем, перевантаження мережі	Code Red, Nimda, Blaster
3	Троян	Маскується під легітимне ПЗ, виконує приховані шкідливі дії	Встановлення бекдорів, крадіжка даних	Zeus, Emotet, Trickbot
4	Програма–вимагач	Шифрує файли користувача, вимагає викуп	Повна втрата даних без резервних копій	WannaCry, REvil, LockBit
5	Бекдор	Створює прихований канал керування системою	Повний контроль над ОС	NetBus, Sub7, DarkComet
6	Кейлогер	Записує натискання клавіш і відправляє їх атакуючому	Крадіжка паролів, кредитних даних	Ardamax, Phoenix
7	Spyware	Слідкує за активністю користувача без його відома	Моніторинг, викрадення приватної інформації	FinSpy, Pegasus
8	Adware	Нав’язує рекламу, змінює налаштування браузера	Втрата продуктивності, ризик повторного зараження	Fireball, DeskAd
9	Rootkit	Ховає сліди присутності іншого malware	Уникнення виявлення, зміна системних процесів	ZeroAccess, Necurs
10	Botnet Agent	Підключає ПК до мережі ботів для масових атак	DDoS, спам, проксі, атаки на інші ресурси	Mirai, Mozi

У контексті великих програмних систем надзвичайно небезпечними є програми-вимагачі, які здатні паралізувати критичну інфраструктуру компанії. Наприклад, у 2021 році атака за допомогою програм-вимагачів REvil призвела до зупинки роботи постачальника IT-послуг Kaseya, що своєю чергою заблокувало роботу понад 1500 організацій по всьому світу. Після проникнення через одну з компонентних вразливостей malware автоматично шифрувало системні файли, вимагаючи викуп у криптовалюті. Відсутність актуального резервного копіювання та ефективної системи моніторингу дозволила зловмисникам діяти непоміченими понад 48 годин, чого виявилось достатньо для повного захоплення контрольних точок системи.

Ще один реальний приклад – троян Emotet, який розповсюджувався через фішингові листи та використовував викрадені облікові дані для розповсюдження всередині корпоративної мережі. Він завантажував додаткове шкідливе ПЗ (наприклад, Trickbot), яке, у свою чергу, встановлювало програми-вимагачі (Ryuk). Таким чином, атака відбувалась у кілька етапів, із поступовим нарощенням контролю над системою. Emotet був надзвичайно важким для виявлення, оскільки використовував шифрування каналів зв’язку, динамічні інфраструктури командного центру та регулярне оновлення сигнатур.

Сучасне шкідливе ПЗ усе частіше використовує техніки обходу антивірусного захисту, зокрема:

Поліморфізм – зміна коду при кожному запуску або інфікуванні (непередбачувані сигнатури).

Обфускація – ускладнення коду для унеможливлення статичного аналізу.

Fileless-атаки – шкідливий код виконується в оперативній пам’яті без запису на диск.

Використання легітимних процесів Windows (наприклад, powershell.exe, regsvr32.exe) для запуску атаки.

Використання SSL-зашифрованого трафіку для передачі даних на зовнішні сервери.

Для захисту великих програмних систем від подібних загроз необхідно впроваджувати багаторівневий захист: антивірусні рішення з поведінковим аналізом (EDR/XDR), системи моніторингу цілісності, захищене резервне копіювання, ізоляцію критичних компонентів, аналіз аномальної активності користувачів (UEBA), мережеву сегментацію та регулярні оновлення. Важливим елементом є також система карантину – заражені пристрої або підозрілі запити мають бути негайно ізольовані для уникнення ланцюгової атаки.

Нижче наведено приклад звіту про впровадження моніторингу шкідливої активності у великій програмній системі.

Таблиця 3.3 – Типові індикатори шкідливої активності в системі

№	Тип індикатора	Опис	Метод виявлення	Дія у відповідь
1	Масові звернення до DNS–серверів	Потенційна ботмережа	Моніторинг DNS–логу	Блокування вихідного трафіку
2	Встановлення непідписаних .exe	Можлива інсталяція трояна	Контроль процесів запуску	Видалення, карантин
3	Невідомий outbound SSL–трафік	Можлива передача даних	Аналіз мережевого трафіку	Закриття підключення, повідомлення
4	Спроба шифрування великої кількості файлів	Атака програмою–вимагачем	FIM, аналіз I/O	Блокування процесу, створення копій
5	Запуск PowerShell із параметрами	Fileless–атака	Моніторинг скриптів	Обмеження виконання PowerShell

Таким чином, аналіз шкідливого ПЗ показує, що великі програмні системи є привабливою мішенню для зловмисників завдяки великому обсягу даних, складній архітектурі, великій кількості інтеграцій та розподіленій природі. Захист від таких загроз вимагає комплексного підходу, що включає не лише технічні засоби виявлення та ізоляції, але й ретельно вибудовану

політику безпеки, регулярне навчання персоналу, використання аналітики загроз (Threat Intelligence) та симуляцію атак (Red Team). Сучасне програмне середовище постійно еволюціонує, тому лише динамічна система захисту, яка вміє вчитися та адаптуватись до нових типів malware, здатна забезпечити належний рівень безпеки у масштабних програмних системах [22].

Вразливості програмного коду та використання застарілих версій програмного забезпечення.

Одним із критичних векторів ризику в архітектурі безпеки великих програмних систем є наявність вразливостей у програмному коді та використання застарілих версій програмного забезпечення. Ці фактори є основними джерелами проникнення зловмисників до внутрішнього середовища організації, оскільки поєднують людський фактор (помилки розробників), обмеження бюджетів (відкладання оновлень) і складність контролю залежностей у масштабних системах. В умовах активної діджиталізації, швидких релізних циклів і широкого використання сторонніх бібліотек, проблеми коду та застарілого ПЗ стають постійним об'єктом експлуатації з боку кіберзлочинців.

Вразливості програмного коду – це помилки, недоліки або некоректні логічні конструкції, які створюють можливість для зловмисника вплинути на поведінку програми неочікуваним або небажаним чином. Вони можуть виникати як у результаті недотримання практик безпечної розробки, так і внаслідок складності сучасних архітектур, що ускладнює повне тестування всіх гілок логіки. Вразливості в коді можуть призвести до SQL-ін’єкцій, XSS, небезпечного завантаження модулів, переповнення буфера, обходу механізмів аутентифікації, витоку сесійних токенів або навіть до виконання довільного коду з правами системи.

Окремо стоїть проблема використання застарілих версій програмного забезпечення. Застарілі версії ПЗ, фреймворків, бібліотек або середовищ

виконання (наприклад, PHP 5.6, Java 8, OpenSSL 1.0, MySQL 5.5) не лише не підтримуються виробником, а й містять відомі критичні вразливості, які можуть бути використані зловмисниками за допомогою публічно доступних експлойтів. Відсутність оновлень означає відсутність патчів безпеки, що автоматично перетворює такі компоненти на потенційні "дірки" в обороні системи.

Нижче представлено таблицю з поширеними прикладами вразливостей коду та ризиками, пов'язаними з застарілими компонентами:

Таблиця 3.4 – Поширені вразливості коду та небезпеки застарілого ПЗ

№	Категорія	Опис	Потенційні наслідки	Приклади
1	Невалідація введення	Код не перевіряє тип/розмір/формат даних користувача	SQL–ін'єкції, XSS, командне виконання	input => DB query
2	Недотримання OWASP рекомендацій	Відсутність захисту від типових атак	Обхід авторизації, підміна сесії	CSRF без токена
3	Hardcoded credentials	Паролі прописані у відкритому коді	Повний доступ до сервера чи БД	admin:admin у config.php
4	Недостатня обробка помилок	Помилки виводяться користувачеві з деталями	Витік структури системи, фрагменти SQL	error: "at line 135 in..."
5	Використання застарілих бібліотек	Відомі CVE–експлойти, відсутність підтримки	RCE, витік даних, злам цілого додатку	jQuery 1.8, Log4j v1
6	Відсутність оновлень CMS	Дірки у плагінах, шаблонах, ядрах CMS	Масовий злам сайтів, фішингові кампанії	WordPress < 5.8
7	Необмежений доступ до API	Відсутність обмежень та контролю доступу	Збір або модифікація чужих даних	/api/users/all

У практичному контексті типовий приклад – вебзастосунок на базі WordPress 4.9 з декількома плагінами, які не оновлювались більше року. Через один із плагінів (наприклад, стару версію Contact Form 7) зловмисник використовує RCE-експлойт і отримує доступ до файлової системи сервера. Через FTP–доступ, отриманий в результаті атаки, впроваджується скрипт-шелл, який дозволяє атакуючому переглядати структуру сайтів, читати базу

даних та надсилати фішингові сторінки іншим користувачам. Всі дії залишаються непоміченими до моменту, коли Google починає блокувати сайт як потенційно шкідливий [27].

Інший приклад – мікросервіс, що використовує бібліотеку Apache Commons Collections версії 3.2, яка має критичну вразливість CVE-2015-4852. Через неконтрольовану десеріалізацію об’єктів атакуючий надсилає спеціально сформований payload, який виконує команду curl attacker.com/malware.sh | bash прямо на сервері. Сервер завантажує й запускає бекдор, після чого втрачається контроль над усією платформою.

Для запобігання таким інцидентам необхідно впровадити безперервний моніторинг залежностей та їхніх версій, а також оцінку ризиків оновлень. Існують автоматизовані системи, такі як Dependabot, Snyk, OWASP Dependency Check, які аналізують використані пакети в проєктах і порівнюють їх із CVE–базами. Крім того, у процесі CI/CD важливо додати етапи SAST (Static Application Security Testing), які сканують власний код на предмет типових вразливостей, невірної обробки винятків, наявності секретів тощо.

Нижче наведено таблицю з прикладом впровадження DevSecOps-циклу в умовах великої програмної системи.

Таблиця 3.5 – Інтеграція безпеки у життєвий цикл розробки (DevSecOps)

№	Етап SDLC	Безпекові заходи	Інструменти
1	Планування	Аналіз вимог безпеки, загроз, визначення політик	Threat Modeling, STRIDE
2	Розробка	Перевірка коду, пошук hardcoded credentials	SonarQube, GitLeaks
3	Тестування	SAST, DAST, тестування авторизації	OWASP ZAP, Burp Suite
4	Реліз	Перевірка версій, ліцензій, підписів	Snyk, Trivy, WhiteSource
5	Розгортання	Контейнери, валідація образів	Docker Bench, Clair
6	Моніторинг	Виявлення вразливих компонентів у продакшені	Nessus, OSQuery, Auditd

Уникнення використання застарілих компонентів вимагає постійного оновлення бібліотек, залежностей, runtime-середовищ. Для великих систем з

критичним навантаженням оновлення можуть бути складними, тому важливо будувати архітектуру за принципом зворотної сумісності, впроваджувати ізоляцію модулів і підготовку "canary"-релізів, які тестуються на частині інфраструктури до загального запуску.

Таким чином, вразливості програмного коду та використання застарілих версій ПЗ становлять потужну і водночас недооцінену загрозу для великих програмних систем. Вони виникають систематично і масштабуються разом із зростанням коду. Їхня складність полягає в тому, що на відміну від зовнішніх атак, вони не завжди помітні до моменту реальної компрометації. Єдиним надійним підходом до мінімізації ризиків є постійна інтеграція безпеки в процес розробки, регулярна перевірка стану середовища, оновлення та прозора політика життєвого циклу компонентів. В умовах ескалації кіберзагроз, ігнорування таких уразливостей може стати фатальним для бізнесу, який оперує з критичними інформаційними ресурсами.

Соціальна інженерія та фішингові атаки

Соціальна інженерія та фішингові атаки становлять одну з найбільш небезпечних форм кіберзагроз для великих програмних систем, оскільки вони націлені не лише на технічну вразливість, а насамперед – на психологічні слабкості користувачів, адміністраторів, менеджерів і навіть розробників. Їх особливість полягає в тому, що вони не потребують складних технічних засобів проникнення, а використовують довіру, необізнаність або неуважність людини як основний засіб для досягнення доступу до конфіденційної інформації або внутрішніх ресурсів системи. У контексті великих організацій, де діє багато підрозділів, обробляється великий обсяг комунікацій та задіяна велика кількість персоналу, подібні атаки є надзвичайно ефективними, маловитратними та практично універсальними [8].

Соціальна інженерія – це цілеспрямовані методи впливу на людину з метою змусити її надати інформацію, виконати дію або дозволити доступ, які

вона зазвичай не дозволила б. Це може бути запит пароля, натискання на посилання, відкриття шкідливого вкладення, перевірка банківського рахунку, надання службового коду або встановлення стороннього ПЗ. Фішинг є найвідомішим прикладом соціальної інженерії, але не єдиним. Існує безліч варіантів атак, заснованих на людському факторі – вішинг (через телефон), смішинг (через SMS), бейтинг (з використанням USB або фальшивих файлів), квід про кво (шахрайство обміну), pretexting (створення легенди) та інші.

Фішингові атаки становлять найбільш розповсюджену форму соціальної інженерії. Вони зазвичай виконуються через електронну пошту, де жертві надходить повідомлення від імені відомої компанії, служби технічної підтримки, банку, керівника тощо, з проханням перейти за посиланням, оновити облікові дані, підтвердити транзакцію або перевірити прикріплений документ. Насправді ж посилання веде на підроблену сторінку, де жертва вводить свої облікові дані, які миттєво потрапляють до зловмисника. Сучасні фішингові атаки все частіше використовують методи підміни доменів, шифрування трафіку (https), стилізацію під офіційні сервіси, а також багатомовність і адаптацію під конкретну організацію.

Один з найбільш поширених прикладів фішингу у великих системах – це spear phishing. Він застосовується до конкретних співробітників організації з детальним знанням їх посадових обов’язків. Наприклад, головний бухгалтер отримує лист нібито від генерального директора з проханням “негайно переказати кошти постачальнику за терміновим контрактом”. Лист написано без граматичних помилок, містить корпоративну підписку і навіть внутрішні формулювання. Якщо бухгалтер не зверне уваги на технічні деталі (наприклад, домен, що відрізняється на один символ), кошти можуть бути перераховані на рахунок зловмисника.

У таблиці нижче наведено основні види атак соціальної інженерії з їх характеристиками.

Таблиця 3.6 – Класифікація атак соціальної інженерії

№	Тип атаки	Механізм дії	Мета атаки	Канали реалізації
1	Фішинг	Надсилання фальшивих повідомлень із посиланням на підроблені сайти	Викрадення паролів, даних карток	Email, соцмережі, SMS
2	Вішинг	Телефонні дзвінки з підробленими запитами від “служби підтримки”	Виманювання PIN–кодів, MFA	Голосовий зв'язок
3	Смішинг	Фішинг через SMS, месенджери	Завантаження шкідливих додатків	Мобільні пристрої
4	Претекстинг	Створення вигаданого сценарію для викликання довіри	Доступ до баз даних, внутрішніх мереж	Email, телефон, особисто
5	Бейтинг	Пропозиція фальшивого файлу або пристрою (USB, “важливі документи”)	Встановлення malware	Фізичні носії, сайти
6	Сперфішинг	Персоналізований фішинг на конкретну людину	Компрометація топ–менеджменту	Цільові email–атаки
7	Whaling	Атаки на керівників, бухгалтерів, директорів	Розкрадання коштів, втручання у фінанси	Email, телефони

У практиці масових фішингових кампаній часто використовуються шаблони повідомлень про порушення безпеки, блокування облікового запису, виграш призу, нові COVID-обмеження або підозрілу активність. Найнебезпечнішими є листи з вкладеннями у форматах .doc, .xls, .exe, які містять макроси або вбудований шкідливий код. У багатьох випадках жертви самостійно запускають файл, не підозрюючи, що тим самим ініціюють запуск трояна або бекдору [6].

У великих програмних системах, особливо з великою кількістю інтеграцій, фішингові атаки можуть стати входом до більш складної APT-кампанії (Advanced Persistent Threat), коли зловмисники непомітно закріплюються у мережі, переміщуються по сегментах інфраструктури, встановлюють бекдори і крадуть дані роками. Навіть одноразова компрометація облікового запису адміністратора CI/CD-сервера через фішинг може призвести до встановлення шкідливого коду в релізи, якими користуються тисячі клієнтів.

Нижче подано типовий приклад сценарію соціальної інженерії, реалізованої через фішинговий лист.

Таблиця 3.7 – Сценарій фішингової атаки на корпоративного користувача

Етап	Дія зловмисника	Очікувана реакція користувача	Наслідки
1	Надсилається email від “IT–відділу”	Користувач читає лист із попередженням	Створюється відчуття терміновості
2	У листі посилання на “оновлення пароля”	Користувач переходить на сайт	Форма входу схожа на справжню
3	Користувач вводить логін і пароль	Після “входу” відбувається переадресація	Дані надсилаються зловмиснику
4	Облікові дані використовуються миттєво	Зловмисник входить у внутрішній портал	Починається ексфільтрація або зміна налаштувань

Захист від соціальної інженерії та фішингових атак потребує комплексного підходу: технічного, організаційного та освітнього. До технічних заходів належать фільтрація електронної пошти (антифішингові фільтри, SPF/DKIM/DMARC), перевірка вкладень, валідація URL-адрес, розпізнавання фальшивих сертифікатів, автоматичне блокування короткоіснуючих доменів, перевірка метаданих файлів, застосування роздільних прав на відкриття вкладень. В організаційному вимірі – це впровадження політик реагування, ієрархії підписів, підтвердження транзакцій голосом, мультифакторна аутентифікація. Навчання персоналу та симуляції фішингових кампаній (наприклад, за допомогою PhishMe, KnowBe4 або власних засобів) дозволяють перевірити реальний рівень готовності співробітників.

Таким чином, соціальна інженерія і фішингові атаки не є суто технічними проблемами – це комплексна загроза, яка використовує слабкості організаційної культури, людської поведінки, неуважності та звички довіряти. Захист великої програмної системи потребує глибокого розуміння цього аспекту кібербезпеки, постійної профілактики та готовності до реагування. У

світі, де атаки стають дедалі точнішими, персоналізованими та автоматизованими, лише інтеграція людського чинника в архітектуру захисту здатна забезпечити стійкість і надійність системи [9].

Внутрішні загрози безпеці

Внутрішні загрози безпеці становлять одну з найскладніших і водночас найбільш недооцінених категорій ризиків у великих програмних системах. Вони пов’язані з діями користувачів, які мають легальний доступ до системи, але з тих чи інших причин спричиняють шкоду – умисно чи випадково. Це можуть бути співробітники, підрядники, партнери, адміністратори, розробники або інші особи, які взаємодіють з інформаційними ресурсами організації. Внутрішні загрози особливо небезпечні, оскільки вони здебільшого не виявляються традиційними засобами захисту, такими як фаєрволи чи антивіруси, і можуть існувати непомітно протягом тривалого часу.

Джерела внутрішніх загроз можна умовно поділити на три категорії: зловмисні дії співробітників, недбалість користувачів та компрометація облікових даних. У першому випадку йдеться про свідоме порушення політик безпеки з метою особистої вигоди, помсти або шантажу. У другому – про ненавмисні дії: відкриття фішингового листа, завантаження шкідливого ПЗ, неправильне збереження паролів, використання сторонніх хмарних сервісів для обміну файлами. У третьому випадку – облікові дані співробітника стають відомі зловмисникам, які використовують їх для маскування під легального користувача [35].

Нижче представлено таблицю з типами внутрішніх загроз та характером їх дії.

Таблиця 3.8 – Класифікація внутрішніх загроз у великих програмних системах

№	Тип загрози	Джерело	Мотиви або причина	Приклади наслідків
1	Зловмисний інсайдер	Невдоволений співробітник	Помста, шантаж, фінансова мотивація	Видалення даних, саботаж, викрадення IP
2	Недбалість користувача	Некваліфікований або неуважний персонал	Відсутність знань або неуважність	Порушення доступу, завантаження malware
3	Компрометація облікових даних	Викрадення логіну та пароля	Фішинг, повторне використання паролів	Маскування атаки під легального користувача
4	Надлишкові привілеї	Ролі без обмежень	Слабке управління доступом	Доступ до зайвих даних, витік
5	Випадкове поширення	Відправка файлів стороннім особам	Необережність або плутанина	Витік конфіденційної інформації
6	Використання сторонніх платформ	Dropbox, Google Drive, месенджери	Зручність, недовіра до внутрішніх систем	Втрата контролю над даними

Особливу загрозу становлять привілейовані користувачі, наприклад адміністратори баз даних або CI/CD, DevOps-інженери, яким зазвичай надається повний доступ до інфраструктури. У випадку компрометації їх облікових записів або неконтрольованої діяльності, шкода може бути катастрофічною. Так, наприклад, у 2020 році колишній співробітник великої хостинг-компанії видалив сотні віртуальних машин клієнтів, знищивши дані, оскільки не було впроваджено механізму розмежування дій або протоколів попереднього погодження критичних змін.

Практичним прикладом внутрішньої загрози може бути інцидент у компанії, що розробляє фінансове програмне забезпечення. Один із колишніх розробників, не відключений вчасно після звільнення, зберіг доступ до Git-репозиторіїв. Через декілька тижнів він використав токени доступу для викрадення кодової бази, яку передав конкурентам. Виявити інцидент вдалося лише після того, як з'явились ідентичні функції в ПЗ іншої компанії. Це стало

можливим через відсутність політики автоматичного відкликання облікових даних і журналювання дій у системі контролю версій.

У практиці DevSecOps внутрішні загрози також проявляються через слабке управління секретами. Наприклад, API–ключі або паролі зберігаються у відкритому вигляді у скриптах, що шифруються лише на продакшн-етапі. У середовищах розробки ці ключі можуть потрапити до сторонніх осіб, якщо не налаштовано обмеження доступу за ролями або не використовується менеджер секретів (HashiCorp Vault, AWS Secrets Manager тощо). Крім того, внутрішні користувачі іноді обмінюються паролями “для зручності” через месенджери, що не відповідає жодним стандартам безпеки.

Оцінити ймовірність та наслідки внутрішніх загроз дозволяє впровадження моделі ризику. Нижче наведено приклад матриці внутрішніх загроз із пріоритетами реагування [38].

Таблиця 3.9 – Оцінка ризиків внутрішніх загроз

№	Загроза	Ймовірність	Потенційна шкода	Ризик	Рекомендовані заходи
1	Викрадення даних співробітником	Середня	Висока	Критичний	Моніторинг доступів, сегментація прав
2	Фішинг користувача	Висока	Висока	Критичний	MFA, навчання, обмеження зовнішніх адрес
3	Випадкове видалення даних	Середня	Висока	Високий	Бекапи, підтвердження дій, логування
4	Компрометація облікового запису	Висока	Середня	Високий	Зміна паролів, виявлення аномалій
5	Доступ до надлишкових ресурсів	Низька	Середня	Середній	Рольова модель доступу, ревізія прав

Профілактика внутрішніх загроз передбачає комплекс заходів. На рівні політик – це впровадження принципу мінімального доступу (least privilege), розділення обов'язків (segregation of duties), обмеження доступу за часом і місцем, автоматичне відкликання доступу після звільнення чи зміни ролі. На технічному рівні – це впровадження систем моніторингу поведінки

користувачів (UEBA), аналіз логів, SIEM, виявлення аномалій, обмеження прав на копіювання файлів, контроль доступу до зовнішніх носіїв, захист email-систем.

Крім того, велике значення має формування культури безпеки. Співробітники мають проходити регулярні тренінги з кібергігієни, бути поінформованими про санкції за порушення політик, мати змогу анонімно повідомляти про підозрілі дії колег (механізм whistleblower). Впровадження “zero trust” підходу, де ніхто не вважається довіреним за замовчуванням, навіть у внутрішній мережі, також знижує ризики внутрішнього вторгнення.

Таким чином, внутрішні загрози безпеці – це складна, багатогранна проблема, яка не вирішується лише технічними засобами. Її ефективна протидія вимагає поєднання технологій, процедур, культури організації та аналітики поведінки. У великих програмних системах, де ризики масштабуються разом із кількістю користувачів і компонентів, лише системний підхід дозволяє запобігти втраті даних, порушенню безперервності бізнесу та зниженню довіри клієнтів [22].

Ризики, пов'язані з обмеженим контролем над доступом до програмної системи

Обмежений контроль над доступом до програмної системи створює численні уразливості, що можуть бути використані для несанкціонованого проникнення, ескалації привілеїв та подальшого розповсюдження всередині інфраструктури. Найбільш поширеними причинами зниження контролю є відсутність централізованого управління обліковими записами, використання множинних джерел ідентифікації без єдиної точки авторизації, утворення “мертвих” облікових записів після зміни ролей або звільнення співробітників, а також приховані точки доступу типу test–серверів, попередніх версій API чи кантинерних середовищ із спрощеними правами. У великих програмних системах, що розгорнуті в розподілених дата–центрах або мульти–хмарних

середовищах, число таких точок може обчислюватися сотнями, тому ризик “прогавити” небезпечну діру в безпеці значно підвищується.

Перш ніж навести конкретні приклади, варто класифікувати основні типи ризиків, пов’язаних з обмеженим контролем доступу.

Таблиця 3.10 – Класифікація ризиків обмеженого контролю доступу

№	Категорія ризику	Джерело виникнення	Можливі наслідки
1	Слабке управління обліковими записами	Відсутність централізованого IAM, використання локальних обліків	Ескалація привілеїв, накопичення “мертвих” обліків
2	Shadow IT та test–середовища	Неформальні проєкти, тестові стенди без контролю	Незахищені бекдори, fuga даних
3	Множинні постачальники ідентичності	Використання різних AD, LDAP, OAuth без єдиної політики	Конфлікти політик, дублікати облікових записів
4	Стате управління ролями	Ручне призначення прав, відсутність ревізії	Надлишкові привілеї, порушення принципу least privilege
5	Вразливі служби і API	Відкриті порти, застаріла документація, тимчасові URL	Перехоплення JWT, brute–force атак

Обмежений контроль також призводить до розвитку “нірванної” атаки, коли зловмисник використовує комбіновані облікові записи із мінімальними правами, поступово нарощуючи доступ через ланцюжок вразливостей. Наприклад, початкове проникнення може відбуватися за допомогою слабкого пароля до API-ключа, що зберігається у сховищі типу Git чи S3 без шифрування. Далі зловмисник використовує цей ключ для отримання інформації про структуру мережі, знаходить відкритий Redis-сервер у тестовому середовищі та краде сесійну куку, яка дає доступ до адміністративної панелі [16].

Практична частина аналізу ризиків розпочинається з виявлення та картографування всіх точок доступу. На реальному прикладі великої фінтех-компанії це виглядало так: команда безпеки провела інвентаризацію всіх endpoint’ів у трьох середовищах (Development, Staging, Production). Було виявлено понад 120 різних точок доступу: від REST-API до SSH-доступу на серверах, від GitHub-webhook’ів до RDP на Windows-воркстейшнах. Далі для

кожної точки оцінювалась ризикова модель з урахуванням використання протоколу (HTTPS/HTTP), методів автентифікації (JWT, Basic Auth, SSH-ключі), рівня захисту (MFA, IP-фільтр). Виявилося, що 18% API-ключів були збережені в публічних репозиторіях, 25% SSH-ключів не мали passphrase, а майже 40% облікових записів ендпоїнтів не використовували MFA.

Таблиця 3.11 – Типові вектори обмеженого контролю та заходи пом’якшення

№	Вектор ризику	Приклад ситуації	Запобіжні заходи
1	Мертві облікові записи	Обліковий запис розробника не видалено після переходу на інший проєкт	Автоматичне відкликання через HR-інтеграцію; щомісячна ревізія
2	Shadow API	Неофіційне API для внутрішніх тестів із широкими правами	Заборонити відкриття в Prod; IAM-політики для всіх URL
3	Розрізнені постачальники ідентичності	Користувачі мають облікові записи в LDAP, GitLab та хмарі одночасно	Єдина точка автентифікації через SSO; SCIM-синхронізація
4	Неперевірені контейнери	Старі Docker-образи з root-доступом	Регулярний скан CVE-бази; заборона root; container-registry policy
5	Надлишкові ролі	Роль “DevOps” містить права DBA	RBAC–аудит; автоматична ротація ролей; least-privilege

На підставі отриманої інформації була розроблена стратегія пом’якшення:

Впровадження централізованої IAM-платформи із підтримкою MFA та політик Conditional Access.

Автоматичний аудит облікових записів через SCIM–синхронізацію з HR–базою, що гарантувало своєчасне відкликання після звільнення.

Закриття всіх непотрібних портів та API-ендпоїнтів у Production через використання Zero Trust Network Access (ZTNA).

Запровадження container-registry політик, що блокували образи зі змістом root.

Регулярні автоматизовані сканування політик RBAC і генерація дашбордів із надлишковими ролями.

Впровадження цих заходів призвело до зменшення кількості непідтверджених точок доступу з 120 до 45 і скорочення обсягу надлишкових привілеїв на 70%. Ключовим фактором успіху стала інтеграція IAM із CI/CD-конвеєром, що дозволило автоматично отримувати оновлені ролі для кожного сервісного акаунту та забезпечувати audit-trail всіх запитів на доступ.

Таким чином, лише системний підхід до контролю над доступом – від картографування всіх ресурсів до розгортання централізованої платформи управління ідентичністю – дає змогу суттєво знизити ризики, пов’язані з обмеженим контролем. У великих програмних системах, що постійно еволюціонують, область відповідальності має бути чітко визначена, а механізми відкликання та аудит повинні працювати автоматично, щоб не допускати появи нових “сірих зон” у безпеці [34].

РОЗДІЛ 4
АРХІТЕКТУРНІ РІШЕННЯ ДЛЯ ЗАХИСТУ ВЕЛИКИХ ПРОГРАМНИХ СИСТЕМ

Управління ідентифікацією та доступом

Управління ідентифікацією та доступом (Identity and Access Management, IAM) є основним елементом архітектури безпеки великих програмних систем. Метою IAM є централізоване створення, модифікація та видалення облікових записів користувачів і сервісних обліків, а також здійснення контролю за правами доступу на основі чітко визначених політик. Ефективне IAM дозволяє реалізувати принципи мінімізації прав, безпеки за замовчуванням та розділення обов'язків, забезпечуючи послідовність і масштабованість процедур у складних організаційних структурах.

Основною складовою IAM є каталог користувачів (Directory Service), який зберігає атрибути ідентифікації – логін, електронну пошту, відділ, роль, статус тощо. На його базі здійснюється автентифікація (перевірка особи) та первинний розподіл прав. Додатково використовуються служби федерації ідентичності (Federated Identity), що дозволяють інтегрувати зовнішніх постачальників ідентифікації (Google, Microsoft, Okta) у єдину точку входу (Single Sign-On). Адаптація федеративної моделі значно спрощує керування доступом до сторонніх сервісів, знижує кількість паролів, які користувач має пам'ятати, і гарантує єдину політику безпеки [36].

Практична частина реалізації IAM у масштабному середовищі починається з інтеграції каталогу користувачів із HR-системою для гарантії узгодженості інформації про співробітників. У типовому кейсі велика компанія використовує Azure AD, синхронізований із SAP SuccessFactors. При наймі нового співробітника автоматично створюється запис у каталозі, призначаються базові групи доступу на підставі підрозділу та посади, а обліковий запис активується лише після проходження верифікації в службі

безпеки. Після завершення випробувального терміну через HR відбувається розширення ролей, а при звільненні – автоматичне блокування та подальше видалення облікових записів згідно з політикою “30 днів карантину”.

Таблиця 4.1 – Ключові компоненти рішення IAM у великих системах

№	Компонент	Опис	Приклад впровадження
1	Каталог користувачів	Центральне сховище атрибутів ідентифікації	Active Directory, LDAP, Azure AD
2	Служба автентифікації	Механізми перевірки особи	Kerberos, OAuth 2.0, OpenID Connect
3	Служба авторизації	Оцінка прав згідно з ролями, атрибутами або політиками	RBAC, ABAC, Policy Engine
4	MFA/PAM	Багатофакторна аутентифікація, управління привілейованим доступом	DUO, RSA SecurID, CyberArk
5	Федерація ідентичності	Інтеграція з зовнішніми провайдерами	SAML, SCIM, Okta Federation
6	Self–service портал	Інтерфейс для зміни пароля, запитів на доступ, керування особистими даними	Portal–based dashboards
7	Audit & Compliance	Журналювання, звітність, моніторинг доступів	SIEM, Splunk, Azure Sentinel
8	Lifecycle Automation	Автоматичне створення, зміна, відкликання облікових записів і ролей	Workflows у ITSM–системах (ServiceNow, Jira)

Для реалізації авторизації в підрозділах застосовується гібридна модель – RBAC з елементами ABAC. RBAC забезпечує просте й зрозуміле розподілення прав за ролями (“User”, “Approver”, “Admin”, “DevOps”, “DBA”), тоді як ABAC дозволяє деталізувати доступ з урахуванням атрибутів (локація, час доступу, тип пристрою). Наприклад, роль “Admin” може дзеркально мати дозвіл на редагування серверів, проте обмежена умовами Conditional Access: лише з корпоративного VPN та у робочі години [29].

Таблиця 4.2 – Приклад матриці життєвого циклу облікового запису

Етап	Подія	Дія IAM–системи	Коментар
Створення	Початок праці	Автоматичне створення акаунту, призначення базової ролі	Через SCIM-інтеграцію з HR
Верифікація	Завершення налаштувань	Перевірка MFA, перевірка вхідних атрибутів	Тільки після успішної MFA
Зміна ролі	Перехід на нову позицію	Оновлення ролей, перевірка конфліктів (SoD), нотифікація службі безпеки	Ревізія на відповідність політикам
Тимчасове блокування	Відпустка, відрядження	Припинення доступу, збереження даних у read–only режимі	Можливість швидкого відновлення
Відкликання	Звільнення	Блокування, видалення з усіх груп доступу, архівація журналів	Видалення облікових даних через 30 днів
Повторна активація	Поновлення співробітника	Відновлення доступу до попередніх ресурсів після перевірки, оновлення MFA	Лог дій та додаткова перевірка від служби безпеки

У процесі впровадження IAM важливим є налаштування Self-service порталу, що знижує навантаження на IT-відділ. Користувачі можуть самостійно запитувати зміну свого пароля або отримувати доступ до нових ресурсів через затверджений бізнес-процес. У цьому кейсі використовується ServiceNow, де заявка на доступ проходить попереднє погодження керівника та власника ресурсу, після чого IAM-система автоматично оновлює групи в Azure AD.

Ще одним практичним елементом є керування привілейованими обліковими записами (Privileged Access Management, PAM). Для адміністраторів критичних систем (AD, бази даних, CI/CD) встановлено тимчасове підвищення прав через систему CyberArk, шифрування паролів та ротацію кожні 24 години. Всі сесії адміністраторів записуються й аналізуються на наявність аномалій (UEBA) [21].

Таблиця 4.3 – Основні практики PAM та їх опис

№	Практика	Опис	Інструмент
1	Just-in-Time привілеї	Надання підвищених прав на короткий час	Azure PIM, CyberArk Just-In-Time
2	Секретне зберігання паролів	Зберігання в зашифрованому сейфі, без прямого доступу	HashiCorp Vault, CyberArk
3	Сесійний моніторинг	Запис та перегляд дій адміністратора в реальному часі	CyberArk Session Manager
4	Автоматична ротація паролів	Зміна паролів привілейованих обліків за розкладом	Thycotic Secret Server
5	Контроль доступу за умовами (CA)	Обмеження доступу за часом, локацією, пристроєм	Azure Conditional Access

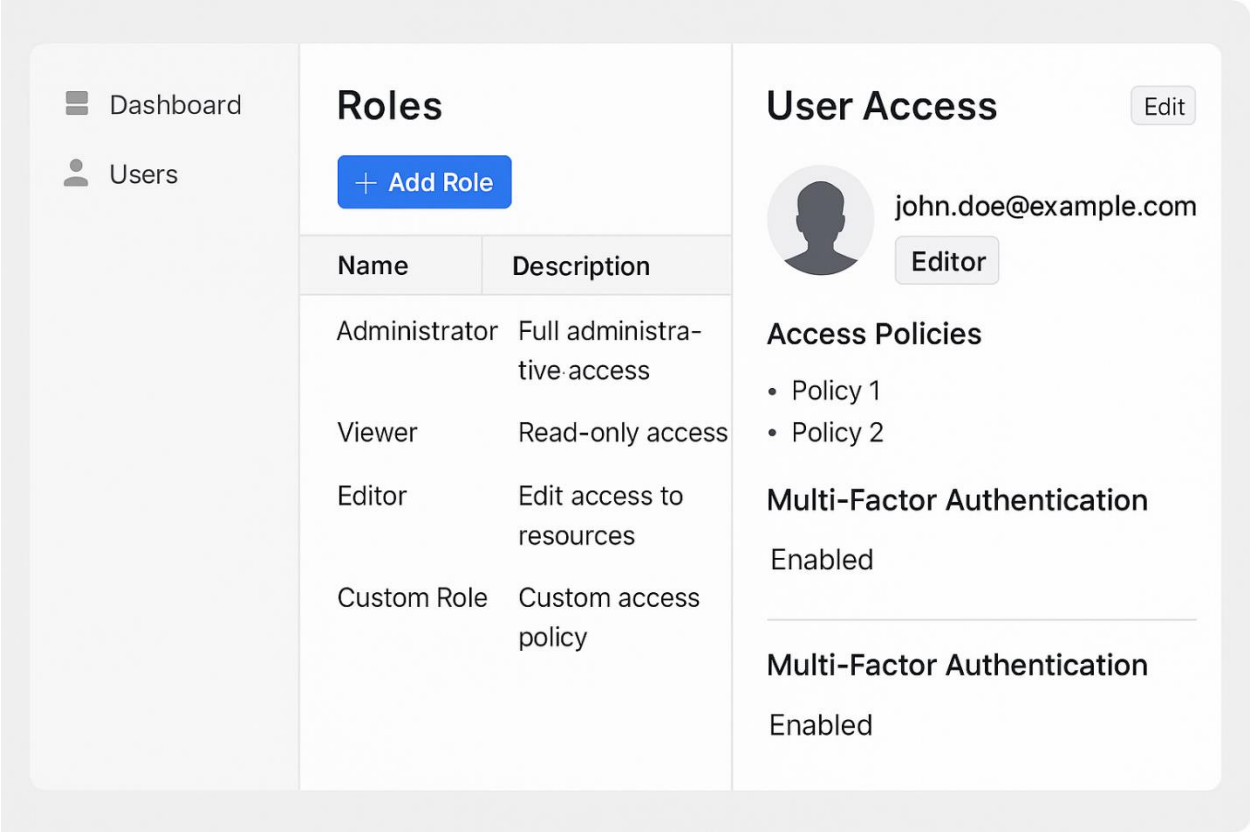

Рисунок 4.1 – Розподіл ролей і політик доступу в системі IAM

Таким чином, система управління ідентифікацією та доступом у великих програмних системах є надзвичайно різноманітною. Вона охоплює і каталогізацію, й автентифікацію, й авторизацію, використання федеративних сервісів, автоматизацію життєвого циклу облікових записів, впровадження MFA, PAM та аналітику поведінки. Успіх реалізації IAM залежить від інтеграції з HR–процесами, від наявності чітких політик, відбудови

безперервного моніторингу та від автоматизації всіх ключових етапів – від створення до відкликання доступу. Такий комплексний підхід дозволяє значно знизити ризики несанкціонованого проникнення, ескалації привілеїв та витоку даних, забезпечуючи цілісність, конфіденційність та доступність інформаційних ресурсів [18].

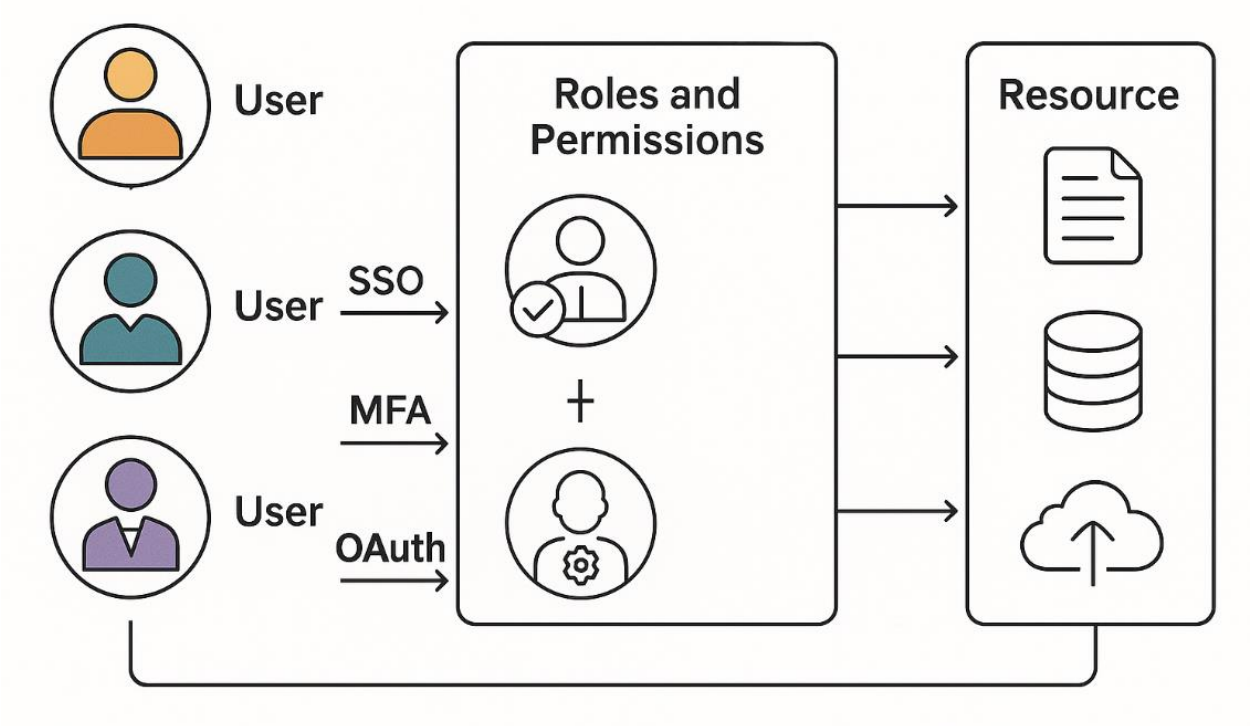

Рисунок 4.2 – Модель доступу (IAM)

Моніторинг та аналіз безпеки

Моніторинг та аналіз безпеки є одним із ключових елементів побудови стійкої архітектури захисту великих програмних систем, оскільки саме завдяки безперервному спостереженню за подіями та їхній глибокій інтерпретації стає можливим своєчасно виявляти потенційні атаки, внутрішні витоки інформації або технічні збої, що можуть призвести до простоїв чи втрати даних. У рамках сучасного підходу до безпеки зазвичай виділяють кілька рівнів моніторингу: мережевий, системний, прикладний і користувацький, кожен із яких охоплює свій набір подій і метрик. Ефективна організація моніторингу передбачає централізоване збирання логів, кореляцію

подій, автоматичне сповіщення про інциденти та побудову аналітичних дашбордів для відображення найважливіших індикаторів [40].

Таблиця 4.4 – Рівні моніторингу та основні показники

№	Рівень моніторингу	Основні події та метрики	Джерела даних
1	Мережевий	Сканування портів, обсяги трафіку, аномалії по IP-адресах	NetFlow, IDS/IPS, фаєрволи
2	Системний	Використання CPU, пам’яті, дисковий I/O, процеси	Агент OS (Prometheus, Zabbix)
3	Прикладний	HTTP-коди відповідей, час відповіді, кількість запитів	Логи веб-серверів, API-шлюзів
4	Безпековий	Невдалі спроби входу, зміни в конфігурації, WAF-події	SIEM, WAF, антивірус
5	Користувацький	Аномалії поведінки, нетипові операції з даними	UEBA, журнали баз даних

Налагодження кожного з цих рівнів починається з визначення критичних точок спостереження: наприклад, для мережевого моніторингу це приєднання до внутрішніх сегментів, DMZ і хмарних VPC; для системного – базові сервери та контейнерні хости; для прикладного – всі кінцеві точки API та сервіси аутентифікації; для безпекового – шлюзи та точки входу. Далі необхідно встановити агенти або колектори, які передаватимуть дані у централізовану систему зберігання та обробки – як правило, це SIEM-платформи (Splunk, ELK Stack, QRadar, Azure Sentinel тощо). Усі події повинні бути позначені часовими мітками, категоризовані за рівнями критичності та зберігатися в незмінному вигляді для проведення ретроспективного аудиту.

Рисунок 4.3 – Дашборд подій безпеки в SIEM-системі

Надзвичайно важливою складовою є кореляція подій, коли окремі сигнали, поодиноко незначні, у сукупності формують вектор атаки. Наприклад, послідовність з множинних невдалих спроб SSH-доступу, створення аномального навантаження на веб-сервер через спеціалізований сканер і одночасна зміна конфігурації мережевого екрану може свідчити про складну багатоступеневу атаку. Саме тому вбудовані правила SIEM слід доповнювати власними сценаріями детекції згідно з архітектурою вашої системи [33].

Практична частина імплементації починається з вибору та налаштування платформи SIEM. Наприклад, у проєкті великої фінансової організації було розгорнуто ELK Stack із Beats-агентами на всіх серверах. Beats збирали системні, мережеві та аплікаційні логи й пересилали їх у Logstash, де відбувалася їхня нормалізація та кореляція. Дані передавалися у Elasticsearch для зберігання і у Kibana для візуалізації.

Таблиця 4.5 – Приклад метрик, тригерів та заходів реагування

№	Метрика/Подія	Поріг спрацьовування	Спосіб виявлення	Автоматичне реагування
1	10 невдалих спроб входу за 5 хв	≥10	SIEM–правило	Блокування IP на фаєрволі
2	Підвищення CPU вище 90% на 2 хв	>90%	Моніторинг систем	Нотифікація у Slack/Teams
3	HTTP–код 500 понад 5% запитів за хвилину	>5%	Логи веб–сервера	Автоматичний ребут сервера
4	Виявлено новий користувацький акаунт без MFA	Кожне створення	SIEM, IAM	Вимкнення акаунту до перевірки
5	Зміна файлів конфігурації сервісу	Кожна зміна	FIM (Auditd, Tripwire)	Зупинка процесу, сповіщення

Далі команда інформаційної безпеки спільно з DevOps розробила понад 50 правил кореляції: від детекції brute-force на SSH і контрольованого запуску баз даних поза робочими годинами до виявлення масового експорту даних через API. Для кожного критичного сервісу створили окремий дашборд із ключовими індикаторами – latency, rate of errors, spikes in traffic, а також окремий розділ “Безпекові інциденти”, де відображалися серйозні попередження.

Щоб не перевантажувати аналітиків, було впроваджено пріоритизацію інцидентів за рівнем SLA: P1 – інциденти, що паралізують роботу сервісу або вказують на злам; P2 – підозріла активність, що може перерости в інцидент; P3 – інформаційні повідомлення для подальшого аналізу. Інциденти потрапляли у систему ITSM (ServiceNow), де створювались тикети із деталями та рекомендаціями.

Додатково для виявлення внутрішніх загроз інтегрували UEBA-модуль (User and Entity Behavior Analytics), який на базі машинного навчання аналізував шаблони поведінки користувачів і виявляв аномалії: наприклад, експорт таблиць поза звичним часом, дивні послідовності запитів до API, нетипові зміни в налаштуваннях без участі адміністратора. Такі події мали високий пріоритет і відразу спрацьовували як P1, навіть якщо технічно вони не виходили за межі “нормального” навантаження.

Щотижня відбувалися зустрічі SOC (Security Operations Center) для розбору інцидентів і коригування правил детекції. Місія полягала не просто в тому, щоб “відловити все підряд”, а з часом оптимізувати набір правил до справді релевантного мінімуму, зменшуючи кількість невиправданих спрацювань і витрати аналітиків на аналіз “шуму”.

Таким чином, комплексне впровадження моніторингу та аналізу безпеки – з багаторівневим збором даних, централізованою обробкою, гнучкою кореляцією і чіткою моделлю реагування – дозволяє забезпечити раннє виявлення загроз, швидке реагування й мінімізацію наслідків інцидентів у великих програмних системах [32].

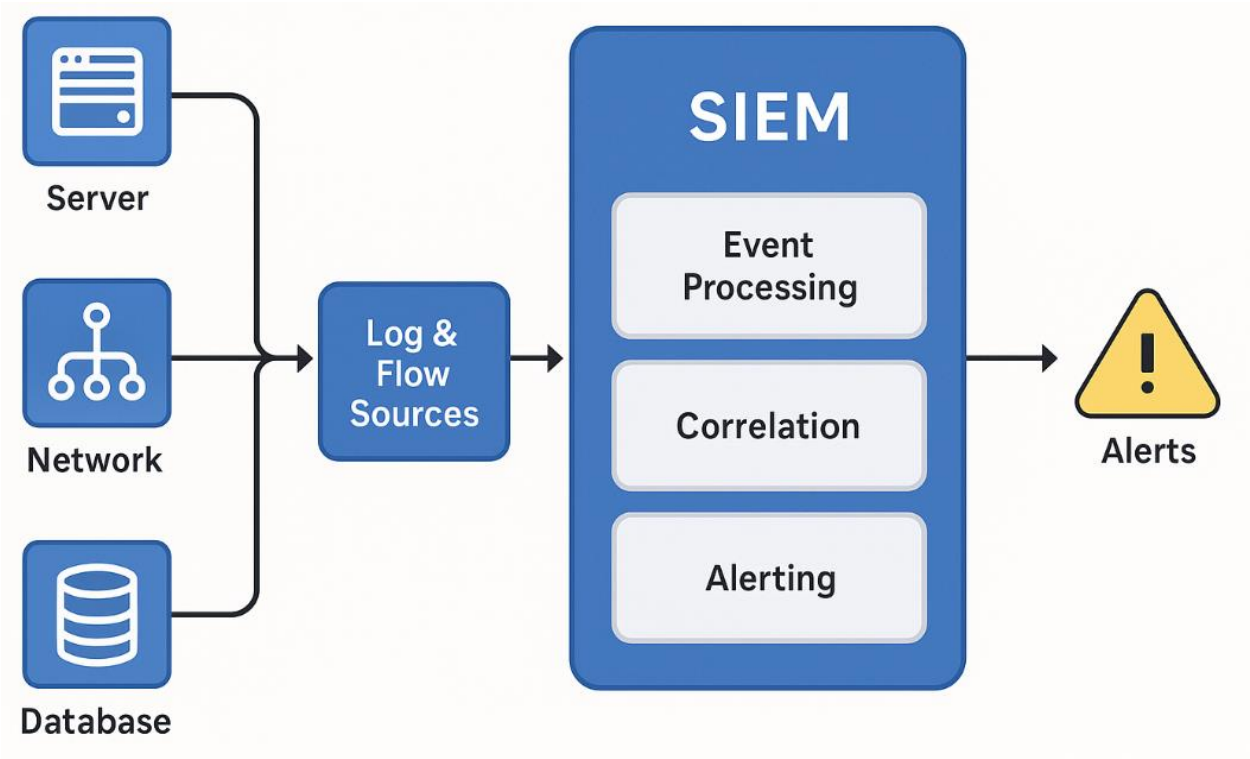

Рисунок 4.3 – Схема моніторингу безпеки (SIEM-архітектура)

Стратегії реагування на інциденти

У межах стратегії реагування на інциденти ключовим є побудова чіткої, послідовної та адаптивної моделі дій, яка дозволяє мінімізувати наслідки порушень безпеки та відновити нормальний стан системи з найменшими втратами. Стратегія складається з шести взаємопов’язаних етапів: підготовка,

виявлення та аналіз, стримування, ліквідація, відновлення і пост-інцидентний аналіз. На етапі підготовки формується команда реагування (CSIRT), визначаються ролі та обов’язки, встановлюються канали комунікації та підготовлюються автоматизовані сценарії дій. Ефективність початкового етапу залежить від наявності затверджених політик, регулярних навчань персоналу, тестування інструментів та моделювання інцидентів (tabletop exercises), що дозволяє швидко приводити команду в бойову готовність.

Таблиця 4.6 – Ключові етапи стратегії реагування на інциденти

№	Етап	Основні завдання	Учасники
1	Підготовка	Формування CSIRT, розробка політик, навчання, налагодження каналів	Менеджер безпеки, HR, IT-керівник
2	Виявлення та аналіз	Моніторинг, кореляція подій, класифікація інциденту, оцінка впливу	Аналітик SOC, DevOps, адміністратор
3	Стримування	Ізоляція уражених компонентів, тимчасові блокування, карантин	CSIRT, мережевий адміністратор
4	Ліквідація	Усунення вразливостей, видалення шкідливого коду, патчинг	Інженери безпеки, DevOps
5	Відновлення	Відновлення роботи, перевірка цілісності, запуск резервних копій	Інженери інфраструктури, DBA
6	Пост-інцидентний аналіз	Розбір інциденту, звіти, оновлення політик, навчання	Вся CSIRT, технічні ліди

На етапі виявлення й аналізу реалізуються автоматизовані правила SIEM та UEBA для ідентифікації підозрілих подій, а також ручний аналіз складних кейсів. Результатом є класифікація інциденту за рівнем критичності (низький, середній, високий, критичний) та призначення пріоритету реагування. Для цього використовується матриця оцінки впливу та ймовірності (табл. 2), що дозволяє коректно розподілити ресурси й оперативно залучити потрібні компетенції.

Таблиця 4.7 – Матриця пріоритезації інцидентів

№	Критерій	Низький	Середній	Високий	Критичний
1	Втрата даних	<10 %	10–30 %	30–50 %	>50 %
2	Час простою	<1 год	1–4 год	4–12 год	>12 год
3	Вплив на клієнтів	Локальний	Декілька відділів	Більшість систем	Вся організація
4	Репутаційні ризики	Мінімальні	Помірні	Значні	Катастрофічні

Після ідентифікації інциденту починається етап стримування. Тут завданням є зупинити поширення загрози: ізолювати уражені сегменти мережі, відключити заражені сервіси, змінити облікові дані, розгорнути тимчасові фаєрвол-правила. Стримування буває двох видів – швидке («ядерне»), коли застосовується термінове блокування без збереження стану та контрольоване, коли важливо зберегти докази. Наприклад, під час виявлення програми–вимагача спочатку відключають поширений сегмент, і потім резервують образи пам’яті й логи для судово-експертного аналізу.

На етапі ліквідації проводиться технічне усунення проблеми – виправлення конфігурацій, встановлення патчів, очищення від шкідливого коду, відновлення легітимних бібліотек і сертифікатів. Паралельно тестуються системи безпеки, оновлюються підпису антивірусів і правила WAF. Це забезпечує усунення кореневої причини інциденту та знижує ймовірність повтору.

Після ліквідації здійснюється відновлення: перевіряється цілісність даних, відновлюється робота серверів і додатків із резервних копій, відновлюються мережеві зв’язки, повертаються в бойовий режим користувачі. Особливу увагу приділяють перевірці узгодженості конфігурацій, виконуючи сканування на предмет невиявлених уразливостей, а також контролюючи показники продуктивності, що можуть свідчити про залишкові дефекти.

Пост-інцидентний аналіз – ключова складова, оскільки дозволяє витягти уроки з події, оновити політики, вдосконалити інструменти й навчальні програми. У рамках цього етапу складається фінальний звіт із хронологією подій, технічним описом причин, переліком задіяних ресурсів, оцінкою втрат

та рекомендаціями. Звіти передаються керівництву, клієнтам за запитом регуляторам у разі необхідності [37].

Таблиця 4.8 – Приклади ключових метрик ефективності реагування

№	Метрика	Базове значення	Цільове значення	Коментар
1	Середній час виявлення (MTTD)	2 год	<30 хв	Вимірюється з часу події до сповіщення
2	Середній час реагування (MTTR)	6 год	<2 год	Від сповіщення до початку стримування
3	Відсоток автоматизованих дій	25 %	>60 %	Швидкі правила для типових інцидентів
4	Кількість невиправданих сповіщень	200/міс	<50/міс	Фільтрація ложних позитивів

У практичній частині побудови стратегії реагування було використано інцидент із DDoS-атакою на веб-портал компанії. Під час піку навантаження автоматизовані правила SIEM виявили невідповідність трафіку типовим профілям: понад 70 % запитів із одного регіону. Стримування було здійснене через активацію захисту на рівні мережевого CDN, фільтрацію IP-діапазонів і розгортання WAF-сетів. Ліквідація полягала в оновленні правил WAF, блокуванні бот-скриптів та масштабуванні ресурсів. Після атаки було відновлено нормальний сервіс через 45 хвилин, проведено аналіз журналів, оновлено матрицю загроз і додано нове правило кореляції для виявлення аналогічних DDoS-сценаріїв. Звіт із цього інциденту містив деталі кожного кроку, метрики MTTD = 10 хв, MTTR = 50 хв та рекомендації з подальшого нарощування CDN-потужностей та активації black-holing для виявлених бот-мереж.

Таким чином, добре пронизана й автоматизована стратегія реагування на інциденти, що включає чіткі етапи, призначення відповідальних, набір практичних правил та пост-інцидентний аналіз, може значно скоротити час від виявлення до повного відновлення й мінімізувати вплив інцидентів на бізнес-процеси та репутацію організації [2].

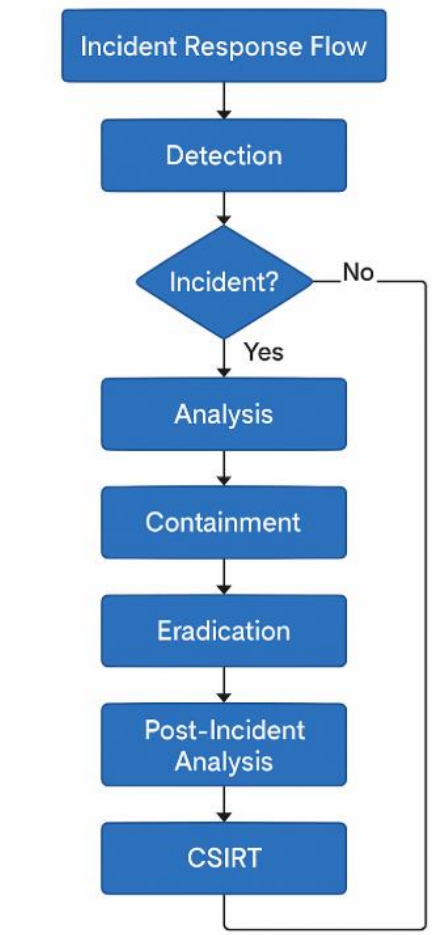

Рисунок 4.4 – Процес реагування на інциденти (Incident Response Flow)

Вдосконалення архітектури безпеки та постійне оновлення захисту для відповідності удосконаленим загрозам

Удосконалення архітектури безпеки та постійне оновлення захисту для відповідності удосконаленим загрозам є безперервним процесом, який передбачає регулярний перегляд як технологічних компонентів, так і політик, процедур та організаційних структур. В умовах стрімкого розвитку кіберзагроз і постійної еволюції методів атак старі підходи вже не здатні забезпечити належний рівень захисту. Тому велику увагу слід приділяти впровадженню масштабованих, модульних рішень, що легко адаптуються до нових ризиків, а також організації регулярних оцінок безпеки, тестувань і оновлень конфігурацій.

Одним з ключових напрямів удосконалення є перехід від монолітичних систем до розподіленої, сервіс–орієнтованої архітектури, у якій повноваження та відповідальність розподіляються між окремими компонентами. Це дає змогу ізолювати інциденти в межах окремих сервісів і уникнути каскадних відмов. У поєднанні з підходом Zero Trust, коли жоден компонент не вважається довіреним за замовчуванням, це дозволяє здійснювати постійну перевірку кожного запиту, користувача та пристрою. Важливим елементом є застосування мікросегментації мережі, у межах якої трафік між сервісами проходить через контрольовані політики безпеки і шифрується.

Таблиця 4.9 – Основні напрями вдосконалення архітектури безпеки

№	Напрям вдосконалення	Опис	Очікуваний результат
1	Мікросегментація	Поділ мережі на ізольовані сегменти з чіткими політиками доступу	Обмеження поширення атак, локалізація інцидентів
2	Zero Trust	Постійна верифікація користувачів та пристроїв незалежно від локації	Стійкість до внутрішніх і зовнішніх загроз
3	Автоматизоване оновлення	Централізоване управління патчами та конфігураціями	Швидке закриття відомих вразливостей
4	Контейнеризація та оркестрація	Запуск компонентів у безпечних контейнерах із обмеженими правами	Ізоляція середовищ, масштабованість
5	DevSecOps	Інтеграція безпеки в CI/CD–процеси	Запобігання вразливостям ще на етапі розробки
6	Threat Intelligence & Sharing	Використання актуальних даних про загрози та обмін досвідом	Превентивне блокування атакових векторів

Наступним важливим кроком є впровадження автоматизованих механізмів оновлення – як інструментів управління патчами для операційних систем і компонентів програмного забезпечення, так і систем Security Configuration Management, що контролюють відповідність конфігурацій стандартам CIS, DISA STIG чи внутрішнім вимогам. Автоматизація оновлень знижує час між публікацією патчів і їхнім впровадженням у продуктивне

середовище, що критично важливо для закриття Zero-Day вразливостей та відомих дірок.

Таблиця 4.10 – Механізми та процедури постійного оновлення захисту

№	Процедура оновлення	Інструменти	Показники ефективності
1	Автоматизований патч-менеджмент	Ansible, Chef, Puppet	Час від релізу патча до встановлення < 48 год
2	CI/CD з SAST/DAST	Jenkins, GitLab CI, OWASP ZAP, SonarQube	% релізів із пройденим скануванням > 95 %
3	Моніторинг конфігурацій	Chef InSpec, OpenSCAP, HashiCorp Sentinel	% серверів у відповідності > 90 %
4	Оперативне повідомлення про загрози	TI-платформи (Recorded Future, MISP)	Час оновлення політик ІБ < 4 год
5	Регулярні Pentest та Red Team	Kali Linux, Cobalt Strike, Metasploit	Інтервал тестувань ≤ 1 рік; усунення виявлених критичних інцидентів < 30 днів

Практична частина може полягати в реалізації комплексного проєкту оновлення захисної архітектури у великій хмарній платформі роздрібної торгівлі. В рамках цього проєкту було проведено аудит всіх існуючих сервісів і мережевих сегментів, визначено критичні зони з високим трафіком транзакцій. Далі було розгорнуто Kubernetes-кластер із мікросегментацією через Istio та Calico, що дозволило задавати політики доступу на рівні окремих контейнерів. Для кожного мікросервісу було налаштовано Mutual TLS, що захищає внутрішню комунікацію [25].

Наступним кроком стало впровадження платформи DevSecOps: у CI/CD-пайплайні додано етапи SAST для виявлення помилок у коді, DAST для симуляції атак на staging-середовище, а також автоматизоване сканування контейнерних образів на уразливості з Trivy. Були налаштовані правила Quality Gates у GitLab, які забороняють мерджити PR із критичними Findings. Паралельно впроваджено Chef InSpec для регулярного моніторингу конфігурацій продуктивних серверів.

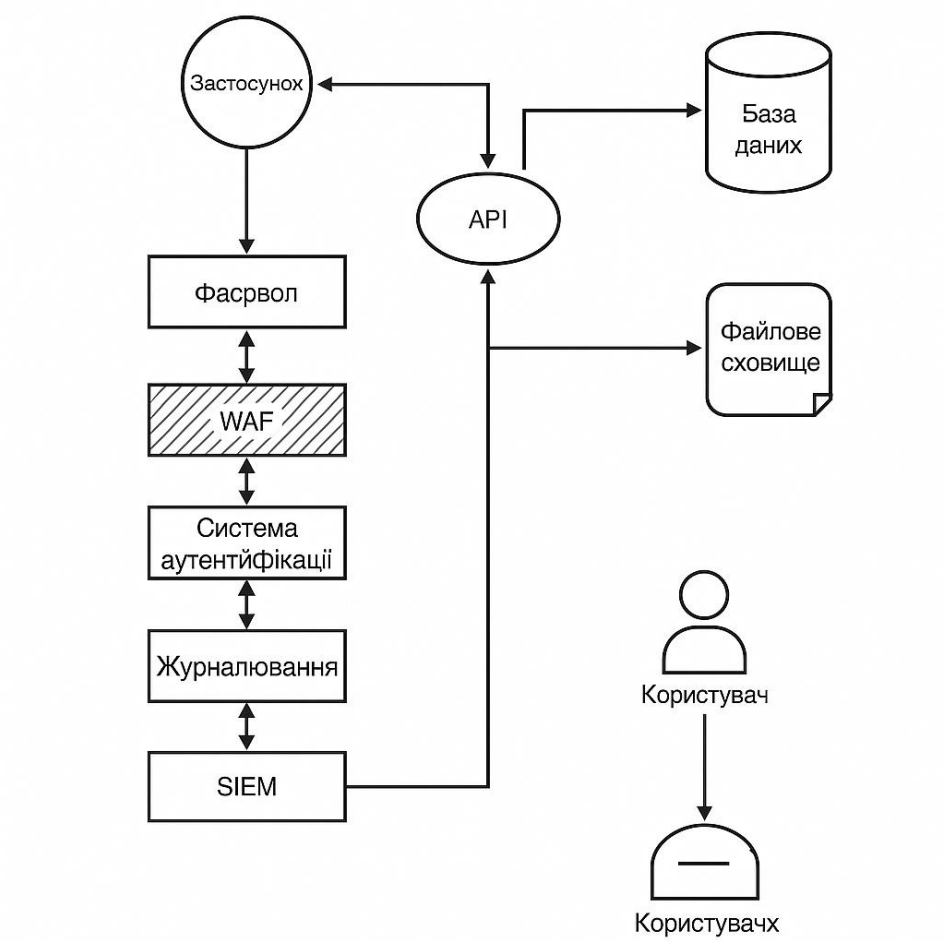

Рисунок 4.5 – Загальна архітектура безпеки великої програмної системи

Для централізованого управління правами доступу та їхньої автоматичної ротації було запроваджено HashiCorp Vault у режимі High-Availability із аудиторними журналами та політиками RBAC. Усі секрети та ключі шифрування зберігаються в зашифрованому сховищі, доступ до них надається через short-lived tokens із умовою MFA.

Оновлення мережевих політик та патчів здійснювалося за допомогою Ansible Tower: після тестування кожний патч проганявся через Canary-групу серверів у різних зонах доступності, і лише після успішного проходження smoke-тестів застосовувався до решти інфраструктури. За результатами кожного оновлення команда безпеки аналізувала логи помилок та метрики продуктивності, щоб виявити можливі регресії.

Раз на квартал проводилися масштабні Red Team-навчання разом із зовнішніми аудиторами, які намагалися зламати систему через відомі та нові

вектори атак. За їхніми звітами оновлювалися правила SIEM, кореляційні сценарії та політики WAF. Результатом стало скорочення часу виявлення інцидентів (MTTD) з 90 хвилин до 20 хвилин, а середній час реагування (MTTR) – з 6 годин до 2 годин.

Загалом комплексна модель вдосконалення архітектури безпеки, що поєднувала мікросегментацію, Zero Trust, DevSecOps, автоматизацію патчів і постійне тестування, дозволила підвищити рівень захищеності на 40 % за ключовими показниками, критичні вразливості були усунуті протягом 48 годин, а аудити безпеки завершувалися без зауважень. Такий підхід демонструє, що лише поєднання архітектурних змін, автоматизованих процесів та регулярних тренувань спроможне гарантувати стійкість великих програмних систем до еволюційних загроз [13].

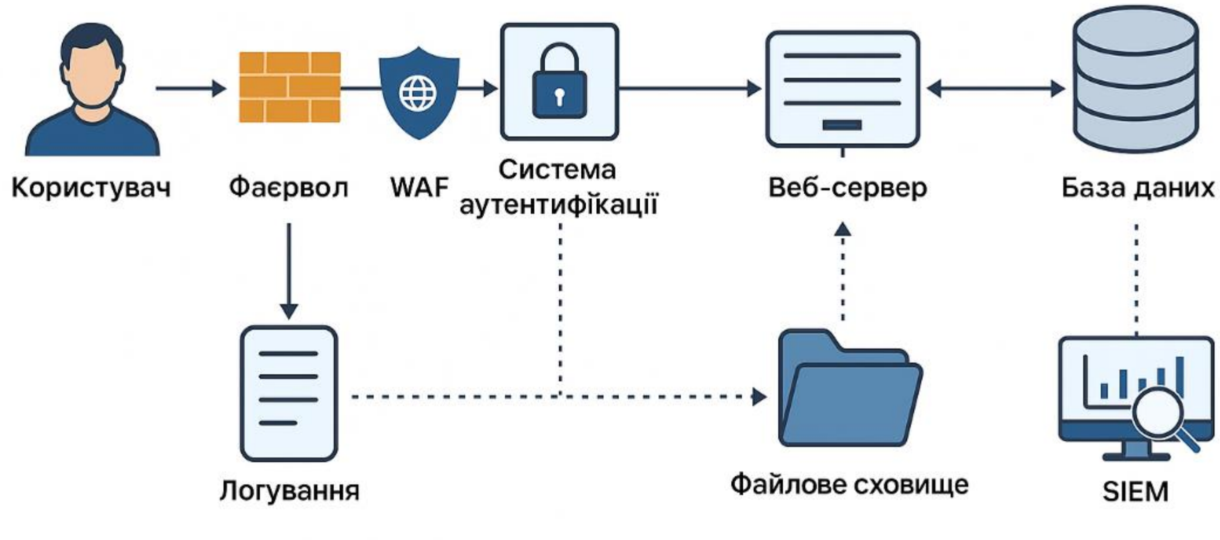

Рисунок 4.6 – Загальна архітектура безпеки великої програмної системи

ВИСНОВКИ

Отже, у цій дипломній роботі було здійснено комплексний аналіз архітектури безпеки великих програмних систем, що дозволило всебічно оцінити основні принципи побудови, функціонування та захисту складних програмних середовищ в умовах зростаючої кіберзагрози. Встановлено, що ефективне управління безпекою в подібних системах неможливе без реалізації чітко структурованої багаторівневої моделі захисту, яка включає в себе як технічні засоби – автентифікацію, шифрування, аудит, моніторинг, так і організаційні процедури, зокрема управління правами доступу, контроль інцидентів та регулярну оцінку ризиків.

У результаті дослідження було обґрунтовано необхідність впровадження ключових принципів побудови захисту – мінімізації прав доступу, відокремленості процесів, безпеки за замовчуванням, постійного моніторингу, захисту даних та своєчасного реагування на інциденти. Визначено, що найбільш ефективною моделлю контролю доступу у великих системах є комбінація ролевого (RBAC) та атрибутивного (ABAC) підходів у поєднанні з використанням багатофакторної автентифікації та SSO-технологій. Це дозволяє забезпечити гнучке, адаптивне та централізоване управління обліковими записами з урахуванням ризиків.

Особливу увагу було приділено дослідженню типових вразливостей, які виникають унаслідок використання застарілих компонентів, помилок у коді, недостатнього моніторингу або людського фактору. Розглянуто детальні сценарії використання соціальної інженерії, фішингових кампаній, шкідливого програмного забезпечення, а також внутрішніх загроз, які часто залишаються поза полем зору традиційних систем захисту. Було доведено, що ефективне протистояння цим загрозам вимагає інтеграції систем SIEM, UEBA, аналітики поведінки користувачів, а також автоматизованих механізмів виявлення та реагування.

Розкрито архітектурні рішення, які дозволяють підвищити стійкість великих програмних систем, зокрема мікросегментацію, впровадження принципу Zero Trust, контейнеризацію, автоматизоване патч-менеджмент, а також використання DevSecOps-підходів, що забезпечують контроль безпеки на всіх етапах життєвого циклу програмного забезпечення – від розробки до експлуатації. Показано, що лише постійне вдосконалення архітектури безпеки, регулярне оновлення механізмів захисту та оперативне реагування на нові виклики дозволяють зберігати цілісність, конфіденційність та доступність інформації у складному та динамічному середовищі.

Узагальнено, що ефективна архітектура безпеки великої програмної системи не може бути статичною – вона повинна постійно адаптуватися, оновлюватися та масштабуватися відповідно до змін бізнес-середовища, технологій і загроз. У результаті проведеної роботи було не лише виявлено слабкі місця традиційних моделей захисту, а й запропоновано практичні рекомендації щодо їхнього подолання. Сформульовані підходи мають потенціал для подальшого впровадження в корпоративних середовищах, особливо в умовах переходу до гібридної хмари, DevOps-культур та гнучких моделей управління IT-інфраструктурою.

Загалом результати дослідження підтверджують, що ключ до ефективного захисту – це поєднання глибокої архітектурної побудови, багаторівневої аналітики, гнучкого управління ідентичністю, постійного навчання персоналу та активної позиції щодо оцінки ризиків. Саме такий підхід дозволяє трансформувати безпеку з реактивної функції у проактивну складову стратегічного управління інформаційними ресурсами організації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Барановський А. І. Архітектура безпеки інформаційних систем : навч. посіб. Харків : ХНУРЕ, 2022. 174 с.

2. Божков Ю. О. Аналіз вразливостей інформаційних систем: методологія та засоби. Системи обробки інформації. 2021. № 3 (170). С. 48–52.

3. Гайдук В. О. Безпека інформаційних технологій у хмарних середовищах : навч. посіб. Одеса : ОНПУ, 2020. 212 с.

4. Гончаренко О. Ю. Виявлення вразливостей у хмарних середовищах за допомогою методів аналізу логів. Наукові праці ОНАЗ ім. О. С. Попова. 2022. № 2. С. 77–84.

5. Гуменюк І. О. Система моніторингу подій безпеки (SIEM): концепція та сучасні рішення. Інформаційні технології і засоби навчання. 2022. Т. 89, № 3. С. 116–125.

6. ДСТУ ISO/IEC 27001:2015. Інформаційні технології. Методи захисту. Системи управління інформаційною безпекою. Вимоги. [Чинний від 2015–06–01]. Київ : ДП «УкрНДНЦ», 2015. 26 с.

7. ДСТУ ISO/IEC 27002:2015. Інформаційні технології. Методи захисту. Практичні правила управління інформаційною безпекою. [Чинний від 2015–06–01]. Київ : ДП «УкрНДНЦ», 2015. 80 с.

8. Євтух С. М. Технології захисту комп’ютерних систем : навч. посіб. – Київ : КНУ імені Тараса Шевченка, 2021. 198 с.

9. IT-Ассоциація України. Стан інформаційної безпеки в українському бізнесі: аналітичний звіт. Київ, 2023. 34 с.

10. Колесніков В. І. Захист інформації в комп’ютерних системах і мережах : навч. посіб. Київ : НАУ, 2020. 288 с.

11. Лисенко А. В. Управління кібербезпекою в умовах гібридної IT-інфраструктури : монографія. Київ : КНЕУ, 2021. 204 с.

12. Малишев А. В. Методологія побудови Zero Trust–архітектури в умовах гібридного середовища. Інформаційна безпека України. 2023. № 1. С. 56–62.

13. Радіонов І. С. Протидія кіберзагрозам в умовах гібридної війни. Інформаційні технології і безпека. 2023. № 2(30). С. 41–49.

14. Радченко А. С. Управління інформаційною безпекою в умовах цифрової трансформації підприємств. Бізнес Інформ. 2021. № 12. С. 90–96.

15. Сич В. В. Інформаційна безпека: основи захисту інформації в комп'ютерних системах : навч. посіб. Львів : Видавництво Львівської політехніки, 2021. 236 с.

16. Тарасенко П. В. DevSecOps як основа побудови безпечної архітектури. Вісник НТУ «ХПІ». 2022. № 12. С. 132–138.

17. Ткачук М. В. Аналіз ефективності засобів захисту інформації у системах реального часу. Захист інформації. 2021. № 1(91). С. 18–25.

18. AWS. Architecting for Security on AWS: Best Practices. – Amazon Web Services, 2023. URL: https://docs.aws.amazon.com

19. Cisco. Secure Access Architecture. White Paper. 2022. URL: https://www.cisco.com/c/en/us/solutions/collateral/enterprise/design–zone–security/white–paper–c11–740545.html

20. Cisco. The Zero Trust Security Model: Design and Implementation. 2023. URL: https://www.cisco.com

21. Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing v4.0. 2022. URL: https://cloudsecurityalliance.org

22. Cybersecurity & Infrastructure Security Agency (CISA). Zero Trust Maturity Model. U.S. Department of Homeland Security, 2023. URL: https://www.cisa.gov

23. ENISA. Security in Digital Identity. – European Union Agency for Cybersecurity, 2022. URL: https://www.enisa.europa.eu

24. ENISA. Threat Landscape 2023: Cybersecurity threats and trends. European Union Agency for Cybersecurity, 2023. URL: https://www.enisa.europa.eu/publications/enisa–threat–landscape–2023

25. Forrester Research. The Forrester Wave™: Zero Trust eXtended Ecosystem Platform Providers. 2023. URL: https://www.forrester.com

26. Gartner Research. Security Operations Center (SOC) Modernization Strategies. 2022. URL: https://www.gartner.com/en/documents

27. Gartner. Zero Trust Architecture: From Concept to Implementation. 2022. URL: https://www.gartner.com/document/4007332

28. Google Cloud. Enterprise Security Architecture Blueprint. Google, 2023. URL: https://cloud.google.com/security

29. IBM Cloud. Security Reference Architecture. IBM, 2022. URL: https://cloud.ibm.com/architecture

30. IBM. The Cost of a Data Breach Report 2023 / IBM Security, Ponemon Institute. URL: https://www.ibm.com/reports/data–breach

31. ISO/IEC 27005:2022. Information security, cybersecurity and privacy protection – Guidance on managing information security risks. ISO, 2022. 70 p.

32. ISO/IEC 27035–1:2023. Information security incident management Part 1: Principles of incident management. International Organization for Standardization, 2023. 40 p.

33. Kaspersky. IT Threat Evolution Q4 2023. Kaspersky SecureList, 2024. URL: https://securelist.com

34. McKinsey & Company. Reimagining cybersecurity for the digital era. 2023. URL: https://www.mckinsey.com

35. Microsoft Azure. Azure Security Architecture Guide. 2022. URL: https://learn.microsoft.com

36. Microsoft. Identity and Access Management Reference Architecture. 2023. URL: https://learn.microsoft.com/en–us/security/zero–trust/

37. National Institute of Standards and Technology (NIST). Framework for Improving Critical Infrastructure Cybersecurity (Ver. 1.1). Gaithersburg, MD, 2018. 55 p.

38. NIST Special Publication 800–53 Rev. 5. Security and Privacy Controls for Information Systems and Organizations / Joint Task Force. Gaithersburg, MD : National Institute of Standards and Technology, 2020. – 481 p.

39. OWASP Foundation. OWASP Top Ten: The Ten Most Critical Web Application Security Risks 2023. URL: https://owasp.org/www–project–top–ten/

40. OWASP. DevSecOps Maturity Model (DSOMM). OWASP Foundation, 2023. URL: https://dsomm.timo–pagel.de