

ЛЕКЦІЯ 14. ЗАХИСТ ІНФОРМАЦІЇ В ОПЕРАЦІЙНИХ СИСТЕМАХ

- Організація входу користувачів у систему
- Керування доступом користувачів до даних у UNIX і Windows
- Аудит подій у системі
- Безпека даних на локальному комп'ютері та у мережі
- Засоби захисту від атак на систему

14.1. Основні завдання забезпечення безпеки

Забезпечення безпеки комп'ютерних систем вимагає виконання комплексу завдань, найважливішими серед яких є виконання процедур аутентифікації, авторизації та аудиту, дотримання конфіденційності, доступності та цілісності даних.

Аутентифікація (authentication) – це процес, за допомогою якого одна сторона (система) засвідчує, що інша сторона (користувач) є тим, за кого себе видає. Під час аутентифікації потрібне свідчення (credentials), що найчастіше складається з інформації, відомої обома сторонам (наприклад, ним може бути пароль). Користувач, що пред'явив коректне свідчення, дістає позитивну відповідь на вимогу аутентифікації.

Після того як аутентифікація відбулась успішно, користувач починає працювати із системою. Тепер йому може знадобитися доступ до різних ресурсів. **Авторизація** (authorization), або *керування доступом* (access control) – це процес, за допомогою якого перевіряють, чи має право користувач після успішної аутентифікації отримати доступ до запитаних ним ресурсів.

Авторизацію здійснюють порівнянням інформації про користувача з інформацією про права доступу, пов'язаною із ресурсом. Зазвичай вона містить тип дії та відомості про користувачів, яким дозволено цю дію виконувати. Якщо користувач має право на запитану дію із цим ресурсом, йому надають можливість її виконати.

Під **аудитом** (auditing) розуміють збирання інформації про різні події у системі, важливі для її безпеки, і збереження цієї інформації у формі, придатній для подальшого аналізу. Подіями можуть бути успішні та неуспішні спроби аутентифікації у системі, спроби отримати доступ до об'єктів тощо. Інформацію звичайно зберігають у спеціальному системному журналі (system log). У деяких системах цей журнал має вигляд текстового файла, інші підтримують його спеціальний формат.

Основним компонентом політики безпеки комп'ютерних систем є дотримання найважливіших характеристик даних.

Конфіденційність (data confidentiality) – можливість приховання даних від стороннього доступу, її зазвичай забезпечують криптографічним захистом даних за допомогою їхнього шифрування.

Цілісність (data integrity) – спроможність захистити дані від вилучення або зміни (випадкової чи навмисної). Технології підтримки цілісності даних також пов'язані із криптографічними методами, вони включають цифрові підписи і коди аутентифікації повідомлень.

Доступність (data availability) – гарантія того, що легітимний користувач після аутентифікації зможе отримати доступ до запитаного ресурсу, якщо він має на це право. Порушення доступності даних називають *відмовою від обслуговування* (denial of service), однієї із цілей політики безпеки є запобігання випадковому або навмисному доведенню системи до такої відмови.

Про ці завдання докладніше йтиметься далі.

14.2. Базові поняття криптографії

Припустимо, що одна особа збирається відіслати іншій особі повідомлення і хоче бути впевненою в тому, що ніхто інший не зможе його прочитати навіть тоді, коли воно буде перехоплене.

У криптографії повідомлення називають *вихідним текстом* (plaintext), зміну вмісту повідомлення так, що воно стає недоступним для сторонніх, – *шифруванням* (encryption), зашифроване повідомлення – *шифрованим текстом* (ciphertext), а процес отримання вихідного тексту із шифрованого – *дешифруванням* (decryption).

Криптографічний алгоритм, який ще називають *шифром* (cipher), – це математична функція, яку використовують для шифрування і дешифрування.

Сучасні криптографічні алгоритми проблему безпеки вирішують виключно за допомогою *ключа* (key). Ключ – це значення, яке використовують для шифрування і дешифрування, при цьому без знання ключа дешифрування має бути технічно неможливим (тобто неможливим за наявності будь-яких доступних ресурсів). **Ключ** має належати до множини зі значною кількістю елементів: її розмір повинен бути таким, щоб забезпечити технічну неможливість підбору ключа повним перебором усіх елементів цієї множини.

Сукупність алгоритму і множини всіх можливих вихідних текстів, шифрованих текстів і ключів називають *криптосистемою*.

Криптографічні алгоритми, засновані на ключі, поділяють на дві основні групи: *симетричні* (або *алгоритми із секретним ключем*) і *алгоритми із відкритим ключем*. Далі докладніше про особливості цих двох груп.

Криптографічний протокол – це послідовність кроків, розроблена для виконання деякої задачі із забезпечення криптографічного захисту даних.

Такий протокол обов'язково включає дві або більше сторін і розглядається як обмін діями між сторонами. Для позначення сторін прийнято використовувати такі імена: *Аліса* – учасник, що робить перший крок; *Боб* – учасник, що робить крок у відповідь.

На окремих етапах протоколів використовують різні криптографічні алгоритми.

Алгоритми із секретним ключем або симетричні алгоритми – це такі алгоритми, для яких ключ для шифрування повідомлення збігається із ключем для його дешифрування. Справедлива формула

$$D_K(E_K(P)) = P,$$

де P – вихідний текст, E_K – шифрування із ключем K , D_K – дешифрування із ключем K .

Наведемо деякі приклади симетричних алгоритмів.

Алгоритм *DES* (Data Encryption Standard) був прийнятий як національний стандарт США 1977 року. Головним його недоліком є довжина ключа (56 біт), що робить DES недостатньо надійним у сучасних умовах (ключ може бути знайдений за скінчений час перебором усіх можливих ключів). Як альтернативу сьогодні використовують *потрійний DES* (послідовне шифрування трьома різними ключами).

У 2001 році було прийнято новий стандарт на симетричний криптографічний алгоритм. Він (як і сам алгоритм) дістав назву *AES* (Advanced Encryption Standard). За довжиною ключ *AES* значно перевищує *DES* (стандарт визначає ключі на 128, 192 і 256 біт), крім того, він відрізняється високою продуктивністю.

Подамо у вигляді протоколу *обмін повідомленнями у разі використання криптографічного алгоритму із секретним ключем*.

1. Аліса і Боб погоджуються, що вони використовуватимуть систему із секретним ключем.
2. Вони доходять згоди щодо спільного ключа.

3. Аліса шифрує вихідний текст ключем і посилає його Бобові.

4. Боб розшифровує повідомлення тим самим ключем.

Другий крок потрібно робити секретно через деякий альтернативний канал передавання даних. Якщо Аліса і Боб не можуть використати такий канал, вони змушені передавати ключ незашифрованим, інакше інша сторона не зможе ним скористатися. Якщо при цьому ключ перехопить зловмисник, то він далі зможе читати всю інформацію, яку передають каналом.

Цю проблему неможливо вирішити, залишаючись у межах традиційних симетричних алгоритмів. Потрібен принципово інший підхід. Він отримав назву *криптографії із відкритим ключем* (public key cryptography).

Алгоритми з відкритим ключем розроблено таким чином, що ключ, використаний для шифрування, відрізняється від ключа для дешифрування. Ці два ключі працюють у парі: текст, який шифрують одним ключем, дешифрують іншим відповідно до формул

$$D_{K_{pri}}(E_{K_{pub}}(P)) = P;$$

$$D_{K_{pub}}(E_{K_{pri}}(P)) = P,$$

де $E_{K_{pub}}$, $D_{K_{pub}}$ – шифрування і дешифрування першим ключем пари, $E_{K_{pri}}$, $D_{K_{pri}}$ – шифрування і дешифрування другим ключем пари. Безпека ґрунтується на тому, що один ключ не може бути отриманий з іншого (принаймні за прийнятний для зловмисника проміжок часу).

Ці алгоритми так називаються тому, що один із цих ключів може бути відкритий для всіх (наприклад, опублікований у пресі), даючи змогу будь-якій особі шифрувати цим ключем, але таке повідомлення може прочитати тільки справжній адресат (що володіє другим ключем).

Ключ, що відкривають для інших осіб для виконання шифрування, називають відкритим ключем (public key), парний до нього ключ для дешифрування – закритим ключем (private key).

Алгоритми із секретним ключем працюють швидше за алгоритми з відкритим ключем, тому, якщо не потрібні специфічні властивості, забезпечувані відкритим ключем (наприклад, не передбачене пересилання даних відкритим каналом зв'язку), достатньо обмежитися алгоритмом із секретним ключем.

Найвідомішим алгоритмом із відкритим ключем є *RSA*.

Обмін повідомленнями у разі використання криптографії з відкритим ключем наведемо у вигляді протоколу.

1. Аліса і Боб погоджуються використовувати систему з відкритим ключем.
2. Боб надсилає Алісі свій відкритий ключ.
3. Аліса шифрує повідомлення відкритим ключем Боба і відсилає його Бобові.
4. Боб розшифровує це повідомлення своїм закритим ключем.

Ця послідовність кроків позбавлена недоліків, властивих для процедури обміну секретним ключем. Перехоплення відкритого ключа зловмисником не спричиняє порушення безпеки каналу, а закритий ключ між сторонами не передають.

На практиці алгоритми із відкритим ключем не можуть цілковито замінити алгоритми із секретним ключем, насамперед через те, що вони працюють значно повільніше, і шифрувати ними великі обсяги даних неефективно. Для того щоб об'єднати переваги двох категорій алгоритмів, використовують *гібридні криптосистеми*, де алгоритми із відкритим ключем використовують для шифрування не самих повідомлень, а ключів симетричних алгоритмів, якими далі (після обміну) шифрують весь канал. Такі ключі називають сесійними ключами, бо їх звичайно створюють за допомогою генераторів випадкових чисел для конкретної сесії.

Протокол роботи гібридної криптосистеми наведемо нижче.

1. Боб відсилає Алісі свій відкритий ключ.
2. Аліса генерує випадковий сесійний ключ, шифрує його відкритим ключем Боба і відсилає Бобові.
3. Боб розшифровує сесійний ключ своїм закритим ключем.
4. Обидві сторони далі обмінюються повідомленнями, зашифрованими сесійним ключем.

Гібридні криптосистеми забезпечують якісний захист ключів і прийнятну продуктивність. Про різні застосування цієї технології йтиметься далі.

Ще одна галузь застосування криптографії із відкритим ключем – **цифрові підписи**. Протокол створення і перевірки цифрового підпису при цьому складається з таких кроків.

1. Аліса шифрує документ своїм закритим ключем, тим самим підписуючи його.
2. Аліса відсилає шифрований документ Бобові.
3. Боб розшифровує документ відкритим ключем Аліси, підтверджуючи цифровий підпис.

Цей протокол забезпечує підтримку основних характеристик цифрових підписів, до яких належить, зокрема, неможливість приховати зміну підписаного документа. У цьому разі спроба змінити документ без використання закритого ключа призводить до того, що документ не дешифрується після перевірки підпису.

Алгоритми із відкритим ключем, однак, неефективні для підписування повідомлень великого обсягу. Щоб підвищити продуктивність, цифрові підписи реалізують із використанням односторонніх хеш-функцій.

Хеш-функція – функція, що приймає на вхід рядок змінної довжини, який називають *вхідним зразком*, а повертає рядок фіксованої (звичайно меншої) довжини – *хеш*. За значенням *односторонньої функції* важко віднайти аргумент. **Одностороння хеш-функція** також працює в одному напрямку: легко отримати хеш із вихідного рядка, але технічно неможливо знайти вхідний образ, із якого походить цей хеш. Односторонній хеш-функції властива свобода від колізій: технічно неможливо створити два вхідних образи з одним і тим самим значенням хеша.

Найрозповсюдженішими односторонніми хеш-функціями є MD5 і SHA-1. Крім цифрових підписів, їх можна використати для розв'язання різних задач, які потребують такого відображення важливої інформації, що заслуговує на довіру.

У разі **використання односторонніх хеш-функцій для цифрових підписів із відкритим ключем** замість документа підписують його хеш (значно менший за обсягом). Протокол набуває такого вигляду.

1. Аліса отримує односторонній хеш документа.
2. Аліса шифрує хеш своїм закритим ключем, тим самим підписуючи документ.
3. Аліса відсилає Бобові документ і зашифрований хеш.
4. Боб розшифровує хеш, переданий Алісою, її відкритим ключем.
5. Боб отримує односторонній хеш переданого Алісою документа.
6. Боб порівнює хеші, отримані під час виконання кроків 4 і 5. Якщо вони збігаються, підпис Аліси можна вважати вірним.

Цей протокол ґрунтується на тому, що підпис хеша можна прирівняти до підпису документа, що впливає із властивості свободи від колізій односторонніх хеш-функцій (неможливо створити два документи, які були б перетворені на один хеш).

Сертифікати

Дотепер ми розглядали обмін повідомленнями між двома сторонами. На практиці значно частіше трапляється ситуація, коли ціла низка сторін домовляються про використання криптографії із відкритим ключем для обміну повідомленнями. У цьому разі доцільно розміщувати відкриті ключі кожної зі сторін у спеціальній базі даних. Протокол відсилання повідомлення набуває такого вигляду.

1. Аліса знаходить відкритий ключ Боба у базі даних.
2. Аліса шифрує повідомлення відкритим ключем Боба і відсилає його Бобові.

Основна проблема при цьому пов'язана із базою даних, у якій зберігають відкриті ключі. Вона повинна мати такі властивості.

- Можливість читати з цієї бази даних може мати будь-який користувач.
- У неї не може записувати дані жоден користувач, за винятком деякої довіреної сторони. У протилежному випадку зловмисник зможе записати у базу свій відкритий ключ поверх ключа легітимного користувача і читати всі повідомлення, адресовані йому.

Водночас, навіть якщо база даних і має такі властивості, зловмисник під час передавання даних може підмінити відкритий ключ Боба своїм ключем. Для того щоб цього уникнути, довірена сторона може поставити цифровий підпис на кожен відкритий ключ.

У реальних застосуваннях довірена сторона (центр сертифікації, Certification Authority, CA) підписує не відкритий ключ, а **сертифікат** – документ, що складається із відкритого ключа та інформації, яка ідентифікує його власника. Після підписання сертифікати заносяться у базу даних. Тепер, коли Аліса отримає із бази даних сертифікат Боба, вона зможе переконатися у справжності його відкритого ключа верифікацією підпису CA.

На формат сертифікатів є стандарти, наприклад X.509.

14.3. Принципи аутентифікації і керування доступом

Аутентифікація надає можливість розрізняти легітимні та нелегітимні спроби доступу до системи. Надійна аутентифікація дає змогу у багатьох випадках обмежити коло потенційних порушників легітимними користувачами системи, спрощуючи цим процедури забезпечення її безпеки.

Свідчення, які вимагаються від користувачів під час аутентифікації, найчастіше зводяться до знання секретної інформації, спільної для користувача і системи (наприклад, пароля). До альтернативних свідчень належать:

- володіння деяким фізичним предметом (наприклад, смарт-картою);
- біометричні параметри (відбитки пальців тощо).

Розрізняють локальну і мережну аутентифікацію. У разі успішної локальної аутентифікації користувач доводить свою легітимність для використання ресурсів однієї комп'ютерної системи (свідчення користувача перевіряють локально), мережна аутентифікація дає змогу користувачу довести легітимність для використання всіх ресурсів мережі (свідчення користувача передають для перевірки на спеціальний сервер із будь-якого комп'ютера мережі).

Для того щоб аутентифікація користувача була можлива, у системі має зберігатись інформація про цього користувача. Таку інформацію називають **обліковим записом** (account). Із ним звичайно пов'язують такі дані:

- ім'я користувача, яке він вказує для входу у систему;
- ідентифікатор користувача, що зазвичай є чисельним значенням, унікальним у межах комп'ютера або групи комп'ютерів (цей ідентифікатор ОС використовує під час аутентифікації і авторизації);

- інформація про пароль користувача;
- інформація про обмеження на вхід користувача у систему (термін легітимності облікового запису, періодичність зміни пароля, години і дні тижня, у які користувач може отримувати доступ у систему тощо);
- інформація про групи, до яких належить цей користувач;
- місце знаходження домашнього каталогу користувача (у якому він може створювати свої файли);
- налаштування сесії користувача (шлях до його командного інтерпретатора тощо).

Інформацію про облікові записи зберігають у базі даних облікових записів (account database). Адміністратор системи може змінювати будь-яку інформацію в цій базі, для інших користувачів звичайно доступна лише зміна їхнього власного пароля.

У сучасних ОС для зручності адміністрування системи користувачі можуть об'єднуватись у *групи*. Користувач може одночасно належати до кількох груп. Під час авторизації доступу до об'єктів перевіряють не тільки права самого користувача, але й права груп, до яких він належить.

Інформацію про групи також зберігають у базі даних облікових записів. Звичайно ОС визначає кілька стандартних груп, які створюють під час її установки, зокрема, групу адміністраторів системи (які можуть виконувати в ній будь-які дії) і групу звичайних користувачів із обмеженим доступом.

Для перевірки пароля немає потреби знати цей пароль, досить уміти відрізнити правильний пароль від неправильного. Тому замість зберігання паролів доцільно зберігати односторонні функції цих паролів. Подивимося, як виглядатиме в даному випадку протокол аутентифікації.

1. Аліса посилає системі свої ім'я і пароль.
2. Система обчислює односторонню функцію від пароля.
3. Система порівнює результат обчислення односторонньої функції зі значенням, що зберігається у базі даних облікових записів.

У результаті зменшуються втрати, які може задати зловмисник, коли отримає доступ до списку паролів, оскільки навіть у цьому разі за односторонньою функцією відновити паролі неможливо. Проте цей підхід не позбавлений **недоліків**.

Якщо зловмисник володіє списком паролів, зашифрованих односторонньою функцією, можлива **словникова атака**. Зловмисник бере набір найпоширеніших паролів, застосовує до них односторонню функцію і зберігає всі зашифровані паролі. Потім він порівнює список зашифрованих паролів із цим файлом (словником) у пошуках збігів.

Один зі способів боротьби із такою атакою пов'язаний із використанням *солі* (salt). **Сіль** – це випадковий рядок S , який додають до пароля перед шифруванням. У список шифрованих паролів заноситься рядок $S + E(S + P)$, де P – пароль, E – функція шифрування, «+» – конкатенація рядків. Якщо кількість можливих значень солі достатньо велика, то це робить словникову атаку значно складнішою, оскільки у словник потрібно вносити результати шифрування паролів із усіма можливими значеннями солі.

Солі потрібно досить багато. Наприклад, стандартний її обсяг, прийнятий в UNIX (12 біт, що дає 4096 можливих значень), є не зовсім достатнім (є словники найуживаніших паролів, об'єднані з усіма значеннями солі).

Більш серйозна проблема, пов'язана із використанням описаного підходу, полягає в тому, що пароль передають мережею незашифрованим, і він може бути перехоплений зловмисником. Один зі способів вирішення цієї проблеми полягає в тому, щоб передавати мережею не паролі, а їх односторонні хеші (дайджести). Цей підхід

називають аутентифікацією за принципом «виклик-відповідь» (challenge-response authentication) або дайджест-аутентифікацією. Протокол такої аутентифікації має такий вигляд.

1. Система зберігає значення односторонньої функції від пароля Аліси $F_1(P_S)$ у базі даних облікових записів.
2. Аліса передає системі своє вхідне ім'я (відкритим текстом) і значення $F_1(P_A)$, обчислене із використанням пароля P_A , який вона ввела.
3. Система генерує випадкове число C , яке називають *викликом* (challenge), і передає його Алісі.
4. Аліса застосовує іншу односторонню функцію до значення виклику. На вхід цієї функції, крім виклику, передають значення $F_1(P_A)$
 $R_A = F_2(F_1(P_A), C)$.
5. Аліса передає системі значення R_A (*відповідь*, response).
6. Система обчислює аналогічне до R_A значення R_S на підставі інформації із бази даних облікових записів
 $R_S = F_2(F_1(P_S), C)$.
7. Якщо значення R_A і R_S отримані системою на кроках 5 і 6, збігаються, аутентифікацію вважають успішною.

Використання випадкового значення виклику в цьому разі зумовлене необхідністю запобігання *атаці відтворенням* (replay attack), під час якої зловмисник перехоплює інформацію, передану Алісою системі для того, щоб пізніше відіслати її самому, прикидаючись Алісою.

Цей протокол був основним підходом до аутентифікації у системах лінії Windows XP до появи Windows 2000 і дотепер підтримується у цих системах (наприклад, для локальної аутентифікації). Його надійність залежить від надійності алгоритму, використаного для односторонньої функції (перехоплення зловмисником дайджесту дасть можливість здійснити на нього словникову атаку). Зазначимо також, що в цьому разі користувач не може бути впевнений, що система насправді є тією, до якої він запитує доступ.

Більш складним протоколом аутентифікації є *протокол Kerberos*. Це розподілена система аутентифікації користувачів із можливістю аутентифікації клієнта і сервера. Протокол Kerberos є основним протоколом мережної аутентифікації у системах лінії Windows XP, починаючи із Windows 2000. Реалізація цього протоколу доступна і для UNIX-систем.

Проблему пересилання пароля мережею можна також розв'язати, використовуючи паролі, дійсні лише один раз під час сесії користувача. Перехоплення такого **одноразового пароля** (one-time password) нічого не дає зловмисникові.

Протокол використання одноразових паролів наведено нижче.

1. Аліса задає випадкове значення R для ініціалізації системи.
2. Система обчислює односторонні функції $F(R)$, $F(F(R))$, $F(F(F(...(R))))$, наприклад $n + 1$ раз. Назвемо ці числа $x_1 \dots x_{n+1}$; із них $x_1 \dots x_n$ система передає Алісі, а сама зберігає x_{n+1} .
3. Входячи в систему, Аліса задає своє ім'я і число x_n . Система обчислює $F(x_n)$ і порівнює результат із x_{n+1} . Якщо значення збігаються, аутентифікацію Аліси вважають успішною, і збережене системою x_{n+1} замінюється на x_n .
4. Для наступної аутентифікації Аліса використає число x_{n-1} , далі – x_{n-2} і т. д. Коли буде використане число x_1 , систему повторно ініціалізують новим числом R .

Такі паролі у сучасних системах можуть реалізовуватися за допомогою смарт-карт – електронних пристроїв, у які вбудований мікропроцесор із засобами генерації відповідних одноразових паролів.

Для реалізації **керування доступом** операційна система має можливість визначати, що за дії і над якими об'єктами має право виконувати той чи інший користувач системи. Звичайний розподіл прав відображають у вигляді *матриці доступу*, у якій рядки відповідають суб'єктам авторизації (користувачам, групам користувачів тощо), стовпці – ресурсам (файлам, пристроям тощо), а на перетині рядка і стовпця зазначені права цього суб'єкта на виконання операцій над даним ресурсом (рис. 14.1).

	Файл a.txt	Файл /bin/bash	Принтер L
Бекас	RW	RWX	W
Дендик	---	RWX	W
USERS	---	R	W

Рис. 14.1. Матриця доступу

Зберігання повної матриці контролю доступу неефективне (вона займатиме багато місця), тому звичайно використовують два базові підходи для її компактного відображення.

1. У разі реалізації *списків контролю доступу* (Access Control Lists, ACL) зберігаються стовпці матриці. Для кожного ресурсу задано список суб'єктів, які можуть використовувати цей ресурс.
2. У разі реалізації *можливостей* (capabilities) зберігаються рядки матриці. Для кожного суб'єкта задано список ресурсів, які йому дозволено використовувати.

Якщо реалізовано ACL, для кожного об'єкта задають список, що визначає, які користувачі можуть виконувати ті чи інші операції із цим об'єктом. Наприклад, для файлових систем такими об'єктами є файли або каталоги. У найзагальнішій формі елементи цього списку – це пари (користувач, набір дій), які називають **елементами контролю доступу** (access control elements, ACE).

Розглянемо деякі проблеми, які потрібно вирішити у разі реалізації списків контролю доступу.

- Розмір списку контролю доступу має бути не надто великим, щоб система могла ефективно ним керувати,
- Права мають бути досить гнучкими, але при цьому не дуже складними. Надмірне ускладнення системи прав звичайно призводить до того, що користувачі починають її «обходити», задаючи спрощені права; у результаті загальна безпека не підвищується, а ще більше знижується.

Вирішення цих проблем призвело до особливостей реалізації списків контролю доступу в різних ОС. Так, загальною концепцією у створенні списків є задавання прав не лише для окремих користувачів, але й для груп користувачів, а також можливість визначити права доступу всіх користувачів системи. У деяких ОС обмежують кількість елементів у списку (наприклад, в UNIX список, як незабаром побачимо, обмежений трьома елементами). Водночас у системах лінії Windows XP можна задавати списки контролю доступу, що складаються з необмеженої кількості елементів (для окремих користувачів, груп, користувачів інших комп'ютерів тощо), і задавати різні комбінації прав – від узагальнених до детальних.

Під час визначення можливостей для кожного користувача задають, до яких файлів він може мати доступ і як саме. При цьому із користувачем пов'язують **список можливостей** (capability list), що складається із пар (об'єкт, набір дій).

Реалізація списків можливостей дає змогу забезпечити не тільки захист, але й задавання імен. Можна зробити так, щоб кожний користувач бачив тільки ті файли, до яких у нього є доступ. Якщо при цьому значенням за замовчуванням є відсутність доступу, користувачі можуть починати роботу в «порожній» системі. Загалом можливості використовують у системах, де потрібно забезпечити більшу безпеку за рахунок деякого зниження зручності роботи.

Можливості можуть бути задані не тільки для користувачів, але й для окремих процесів. У результаті під час запуску кожного процесу можна визначити його права.

14.4. Аутентифікація та керування доступом в UNIX

Перш ніж розглянути реалізацію аутентифікації в UNIX, зупинимось на концепції користувача цієї системи.

Кожному користувачу в UNIX ставлять у відповідність **обліковий запис** (account), що характеризується іменем користувача та ідентифікатором (uid). Як ім'я користувача, так і його ідентифікатор мають бути унікальними в межах усієї системи. Крім цього, з обліковим записом користувача пов'язують його **домашній каталог** (home directory), у який він за замовчуванням може записувати дані. Після входу користувача у систему відбувається перехід у його домашній каталог.

Користувачі об'єднуються в групи. Кожна група характеризується іменем та ідентифікатором (gid).

Загалом процес виконують із правами того користувача, який запустив відповідний виконуваний файл.

Для доступу до ідентифікатора користувача поточного процесу використовують системний виклик `getuid()`, до ідентифікатора основної групи цього користувача – `getgid()`:

```
#include <unistd.h>
printf("uid = %d, gid = %d\n", getuid(), getgid());
```

Користувач із uid, що дорівнює нулю (звичайно його називають root) має в UNIX особливий статус – він може виконувати у системі будь-які дії без обмежень. Такого користувача називають також **суперкористувачем**. Усі інші – *звичайні користувачі*.

Наявність єдиного суперкористувача (принцип «все або нічого») вважають головною слабкістю системи безпеки UNIX, оскільки компрометація єдиного пароля root негайно призводить до того, що зловмисник отримує повний контроль над системою. Крім цього, процеси під час виконання не можуть бути обмежені якоюсь частиною прав суперкористувача. Фактично, процес, якому потрібна лише невелика частина таких прав, змушений виконуватися під керуванням root із повним контролем над системою.

Для **стандартної аутентифікації** UNIX використовують підхід із використанням односторонньої функції від пароля і солі. Такою функцією традиційно був алгоритм DES, яким шифрувався рядок, що складався із нулів, при цьому в ролі ключа використовували значення пароля, об'єднане із сіллю. У сучасних версіях UNIX замість DES часто використовують односторонню хеш-функцію (звичайно MD5).

За вхід користувача у систему відповідають дві утиліти: `getty` – ініціалізує термінал, видає підказку «login:» і приймає ім'я користувача, і `login`, що видає підказку «password:», приймає значення пароля, робить аутентифікацію і починає сесію користувача (переходить у домашній каталог і запускає командний інтерпретатор).

Інформацію про користувачів у UNIX зберігають у файлі `/etc/passwd`, який відіграє роль бази даних облікових записів. Кожному користувачу відповідає один рядок цього файлу, що має таку структуру:

ім'я_користувача:шифрований_пароль:uid:gid:відомості:home_каталог:shell
де: ім'я_користувача – вхідне ім'я цього користувача;

шифрований_пароль – одностороння функція від пароля, обчислена із використанням DES, MD5 тощо (якщо значення цього поля починається з '*', користувач не може інтерактивно входити у систему);

uid і gid – чисельні ідентифікатори користувача та його основної групи (користувач може належати до кількох груп, інформація про це зберігається у файлі `/etc/group`, що розглядатиметься наступним);

відомості – додаткова інформація про користувача (його повне ім'я тощо);

home_каталог – місце знаходження домашнього каталогу користувача;

shell – версія командного інтерпретатора, який запускають для користувача, коли він входить у систему (у Linux це звичайно `/bin/bash`). Наведемо приклад рядка `/etc/passwd`:

bekas:10elXwSGYJE:540:102:Бекас Б.О.:/home/bekas:/bin/bash

Інформацію про групи зберігають у файлі `/etc/group`. Рядок цього файлу має синтаксис:

ім'я_групи:пароль_групи:gid:список_членів

де: gid – чисельний ідентифікатор групи (той самий, що в `/etc/passwd`);

список_членів – список імен усіх користувачів, які входять у групу, через коми.

Наведемо приклад рядка `/etc/group`:

students::102:bryndas,kitura,vyhrest,firman

Для **доступу** із застосувань до інформації, яку зберігають у `/etc/passwd`, можна використати функції, описані в `<pwd.h>`:

// доступ до інформації для заданого uid

```
struct passwd *getpwuid(uid_t uid);
```

// доступ до інформації для заданого імені користувача

```
struct passwd *getpwnam(char *name);
```

Обидві ці функції повертають покажчик на структуру `passwd`. У таблиці 14.1 наведена відповідність її полів елементам рядка файлу `/etc/passwd`.

Таблиця 14.1.

Відповідність полів структури `passwd` елементам рядка файлу `/etc/passwd`

Поле структури <code>passwd</code>	Елемент рядка <code>/etc/passwd</code>
<code>pw_name</code>	ім'я_користувача
<code>pw_passwd</code>	шифрований_пароль
<code>pw_gecos</code>	відомості
<code>pw_dir</code>	home_каталог
<code>pw_shell</code>	shell

Усі наведені поля є рядками символів. Приклад отримання інформації із цього файлу наведено нижче.

```
struct passwd *pwd;
```

```
pwd = getpwuid(getuid());
```

```
printf("ім'я користувача: %s, домашній каталог: %s\n",
```

```
pwd -> pw_name, pwd -> pw_dir);
```

Заздалегідь виділяти пам'ять під структуру passwd не потрібно – її розміщують у статичній пам'яті під час першого виклику однієї із наведених функцій. Наступні виклики використовують ту саму пам'ять, затираючи попередній результат, тому, якщо потрібно зберегти результат виклику, його копіюють в окрему пам'ять.

У традиційних UNIX-системах до файла /etc/passwd мав доступ будь-який користувач ОС. А тому зломисникові достатньо було отримати непривілейований доступ до системи, щоб почати словникову атаку на цей файл.

Для вирішення даної проблеми потрібно заборонити доступ до цього файла для всіх користувачів, крім root. Проте таке рішення не є прийнятним, оскільки в цьому файлі, крім інформації про паролі, містяться імена та ідентифікатори користувачів, розташування їхніх домашніх каталогів та інші дані, необхідні більшості застосувань. У результаті всі програми, яким потрібна така інформація, довелося б запускати із правами суперкористувача.

У сучасних UNIX-системах (зокрема в Linux) застосовують інший підхід – технологію **тіньових паролів** (shadow passwords). При цьому інформацію про паролі переносять із /etc/passwd в окремий тіньовий файл паролів (який зазвичай називають /etc/shadow). До нього може звертатися тільки суперкористувач. Іншу інформацію (імена та ідентифікатори користувачів тощо) зберігають у /etc/passwd, що залишається доступним для всіх користувачів. Тіньові паролі дають змогу значно підвищити надійність схеми аутентифікації системи.

Головною проблемою цього підходу є те, що в разі переходу до використання тіньових паролів усі прикладні програми, які працюють із паролями (login тощо), необхідно змінити та перекомпілювати. Найгнучкішим вирішенням у цьому разі є застосування *підключуваних модулів аутентифікації* (Pluggable Authentication Modules, PAM), які дають можливість під час роботи системи визначати і налагоджувати процедуру аутентифікації для різних застосувань без зміни їхнього коду.

Реалізація керування доступом до файлів у UNIX збереглася, майже не змінившись від ранніх версій системи. Фактично вона є скороченою реалізацією списків контролю доступу.

Наведемо **основні принципи реалізації керування доступом в UNIX**.

1. Усі файли і каталоги в UNIX мають власника (owner), що належить до певної групи (group), і права доступу (permissions).
2. Розрізняють три категорії прав доступу: для читання, записування і виконання. Для звичайних файлів назви прав відповідають їхньому змісту (зазначимо, що ядро UNIX намагатиметься завантажити у пам'ять і виконати будь-який файл, на який у поточного користувача є права на виконання).
3. Для каталогів задають той самий набір прав, але вони відрізняються за змістом від прав для файлів:
 - право читання для каталогу означає можливість отримання списку імен файлів, що містяться в ньому (тобто читання вмісту каталогу як файла);
 - право записування в каталог означає можливість створення в каталозі нових файлів і вилучення наявних (тобто записування в каталог як у файл); зазначимо, що для вилучення файла прав на нього самого можна й не мати – достатньо прав на каталог, де він зберігається;
 - право виконання означає можливість пошуку в каталозі, тобто доступу до окремих файлів.

Із цього випливає, що якщо для каталогу задаються права «тільки для виконання», то доступ до окремих файлів у ньому можна отримати, коли знати їхні імена, список же всіх файлів отримати буде неможливо. Таке задавання прав використовують, аби

сторонні особи не могли випадково виявити файли, які, однак, потребують спільного доступу. З іншого боку, якщо задати права «тільки для читання», можна переглядати список файлів каталогу, але доступ до окремих файлів стане неможливий.

4. Права кожної категорії задають окремо для власника файлу, для користувачів його групи і для всіх інших користувачів – усього дев'ять базових комбінацій прав; можна сказати, що із кожним файлом пов'язаний список контролю доступу із трьох елементів.

У UNIX-системах є стандартна утиліта виведення вмісту каталогу, яку називають

1s. Ось **фрагмент результату виклику** цієї утиліти для каталогу

```
drwxr-xr-x  2 shekvl  osbook  1024 Sep 8 02:26 dir/
-rwxr-x---  1 shekvl  osbook  11370 Sep 3 13:18 test*
-rw-----  1 shekvl  osbook   149 Sep 3 13:41 test.c
```

Першим полем ліворуч є поле атрибутів файлу, формат якого розглянемо докладніше. Перший стовпчик цього поля визначає тип файлу (для каталогів у ньому стоятиме символ `d`, для звичайних файлів – прочерк, для символічних зв'язків – `l`).

Далі йде символічне відображення прав доступу. Воно складається із трьох трійок символів `rwX`. Ці трійки визначають права власника, групи і всіх інших користувачів. У кожній трійці символ `r` означає право на читання, `w` – на записування, `X` – на виконання. Якщо відповідного права немає, то замість символу стоятиме прочерк.

У наведеному списку файлів:

- для каталогу `dir` задані усі права для власника, а також права на читання і виконання для групи та інших користувачів; це означає, що додавання і вилучення файлів дозволене власникові каталогу, інші дії – усім;
- виконуваний файл `test` може бути прочитаний і запущений власником і членами його групи, записаний – тільки власником, усі інші користувачі доступу до нього не мають;
- файл `test.c` може бути прочитаний і записаний тільки його власником, інші користувачі жодних прав на нього не мають.

Для зміни прав доступу до файлу використовують системні виклики `chmod()` (файл задають іменем) і `fchmod()` (файл задають дескриптором), для зміни власника і групи – `chown()` і `fchown()`. Є також утиліти `chmod`, `chown` і `chgrp`. Опишемо особливості використання виклику `chmod()`:

```
int chmod(const char *pathname, int mode);
```

Параметр `mode` визначає права доступу. Вони можуть бути компактно записані у вісімковому числовому відображенні. Елемент списку контролю доступу відображають сумою трьох чисел: перше число – для права на виконання, 2-ге – на записування, 3-тє – на читання. Значення у будь-якій позиції означає: 0 – відсутність прав, 5 – читання і виконання, 6 – читання і записування, 7 – усі права. Увесь список відображають сукупністю трьох чисел (у тому самому порядку, що і для символічного відображення); так, права для `dir` у цьому випадку були б задані як 755, для `test*` – 750, для `test.c` – 600.

Ось приклад використання виклику `chmod()` для задавання прав, що визначають можливість читання і записування для власника, читання для групи і відсутність доступу для інших користувачів:

```
chmod("./test.c", 0640);
```

Як було видно раніше, таке саме числове відображення може бути передане як режим створення файлу у системні виклики `open()`, `mkfifo()` тощо:

```
int fd = open("./myfile.txt", O_RDWR | O_CREAT, 0640);
```

Для отримання прав доступу до заданого файлу необхідно скористатися системними викликами `stat()` або `fstat()`:

```
struct stat attrs;
stat("./myfile.txt", &attrs);
//S_IFMT задає маску, що визначає всю інформацію, крім прав доступу
printf("права доступу: %o\n", attrs.st_mode & ~S_IFMT);
```

Для перевірки того, чи є в поточного процесу права на доступ до заданого файла, використовують системний виклик `access()`:

```
int access(const char *pathname, int mode);
```

де: `pathname` – повний шлях до файла;

`mode` – тип перевірки (R_OK – наявність права на читання, W_OK – на записування, X_OK – на виконання, F_OK – перевірка на наявність файла).

Виклик `access()` повертає 0 за наявності прав, -1 – за їхньої відсутності.

```
if (access("myfile.txt", R_OK) == -1)
printf("немає права на читання файла\n");
```

Описана схема керування доступом проста для розуміння, але багато в чому обмежена. Основне обмеження пов'язане із довжиною списку контролю доступу (3 елементи). Звідси виникають такі проблеми, як неможливість надання конкретному користувачу прав доступу до файла без створення для нього окремої групи (що не можливо зробити без наявності прав адміністратора).

Останнім часом в UNIX-системах усе ширше використовують списки контролю доступу без обмеження кількості елементів (у Linux їхня реалізація стала доступна у ядрі 2.6).

Права доступу для файлів і каталогів у UNIX дають *зможу задавати додаткові біти*. Одним із таких бітів є `setuid`-біт. Для його задавання у числовому зображенні ліворуч потрібно додати 4 і обов'язково задати право виконання для власника (наприклад, задати права як 4755). Символьне відображення в цьому разі має такий вигляд: `rwsr-xr-x`.

`Setuid`-біт задає для файла режим виконання із правами власника. Під час запуску такого файла на виконання створюють `setuid`-процес, для якого `uid` відповідає користувачу, що запустив програму, а `euid` – власникові виконуваного файла. Ядро ОС під час авторизації доступу до файлів завжди використовує `euid`. У результаті маємо, що `setuid`-процес отримує під час доступу до файлів такі самі права, що і власник відповідного виконуваного файлу.

Прикладом використання такого біта є програма `passwd`, що дає змогу користувачу задати пароль на вхід у систему. З одного боку, вона має бути доступна для запуску будь-якому користувачу системи, з іншого – під час зміни пароля їй потрібно перезаписувати файли `/etc/passwd` або `/etc/shadow`, що вимагає виконання із правами `root`. Щоб задовольнити ці вимоги, для виконуваного файла `passwd` власником вказують `root` і задають `setuid`-біт.

Для визначення ефективного ідентифікатора користувача для поточного процесу використовують системний виклик `geteuid()`:

```
// для setuid-програми uid і euid можуть розрізнитися
printf("uid=%d, euid=%d\n", getuid(), geteuid());
```

Застосування, які виконуються із правами `root`, можуть переходити до виконання із правами інших користувачів за допомогою системного виклику `setuid()`, що змінює значення `uid` і `euid` для поточного процесу:

```
struct passwd *pw = getpwnam("nobody");
setuid(pw->pw_uid);
// тепер процес виконується
// з правами користувача nobody
```

Таку технологію, наприклад, використовують у веб-сервері Apache, основний процес якого запускають із правами `root`, після чого він виконує підготовчі дії, а потім створює нащадків для обслуговування запитів, які негайно переходять до виконання із правами непривілейованого користувача.

Залишається з'ясувати, які права задають для нових файлів. Цим керує встановлення прав за замовчуванням. Якщо його не змінювати, каталоги створюватимуть із правами 777 (усі дії дозволені всім), а файли – із правами 666 (усі дії, крім виконання, дозволені всім).

Змінити встановлення прав можна, задаючи спеціальний *параметр umask* за допомогою системного виклику `umask()` або однойменної утиліти. Під час створення нового файла значення цього параметра побітово віднімається від 666 (вісімкового) для звичайних файлів і з 777 – для каталогів; права для файла будуть результатом цього віднімання. Наведемо приклад виклику `umask()`:

```
umask(022);
```

Після виконання цього виклику файли створюються із правами 644, а каталоги – із правами 755. Найчастіше задають значення `umask`, що дорівнюють 022 і 002, задавання 077 спричинятиме створення файлів, закритих для всіх, крім власника.

Ядро Linux дає змогу задавати списки можливостей для процесів, починаючи з версії 2.4. Наявність підтримки можливостей дозволяє обійти обмеження концепції «все або нічого», коли процесу потрібна тільки частина прав суперкористувача і він змушений брати їх усі, ризикуючи нашкодити системі. Тепер такий процес може задати набір можливостей, що точно описує права, які йому потрібні.

14.5. Аутентифікація і керування доступом у Windows XP

Архітектури безпеки Windows XP містять такі основні компоненти.

- *Процес реєстрації користувачів* (logon process, winlogon) обробляє запити користувачів на реєстрацію у системі та приймає дані від них.
- *Менеджер аутентифікації* (Local Security Authority Subsystem, LSASS) безпосередньо проводить аутентифікацію користувача. Цей компонент – центральний у підсистемі безпеки. Крім аутентифікації, він контролює політику аудиту.
- *Менеджер облікових записів* (Security Accounts Manager, SAM) підтримує базу даних облікових записів (базу даних SAM), що містить імена локальних користувачів і груп, а також паролі. Під час перевірки прав користувача SAM взаємодіє із менеджером аутентифікації.
- *Довідковий монітор захисту* (Security Reference Monitor, SRM) перевіряє права користувача на доступ до об'єкта і виконує необхідну дію з об'єктом за наявності цих прав. Це єдиний компонент, що виконується в режимі ядра, він реалізує політику контролю доступу, визначену менеджером аутентифікації. SRM гарантує, що будь-який користувач або процес, що дістав доступ до об'єкта, має всі права на нього.

Ця архітектура реалізована у системах, що не використовують мережної аутентифікації. Для неї інформація про облікові записи має зберігатися не у базі даних SAM, а у спеціальному централізованому сховищі даних – *активному каталозі* (Active Directory). Під час мережної аутентифікації менеджер аутентифікації звертається до служби активного каталогу віддаленого комп'ютера, на якому розташований цей каталог. База даних SAM, однак, є у будь-якій установці ОС лінії Windows XP – у ній зберігають облікові записи для локальної аутентифікації. Подальший виклад торкатиметься локальної аутентифікації.

Windows XP вимагає, щоб кожному користувачу відповідав обліковий запис. Він пов'язаний із *профілем захисту*, який є набором інформації щодо контролю доступу

(ім'я користувача, список його груп, пароль тощо). Профілі захисту зберігають у базі даних SAM і використовують для аутентифікації.

Розглянемо послідовність кроків реєстрації користувача у системі. Процес winlogon очікує введення від користувача. Потік цього процесу виявляє спробу користувача увійти у систему (натискання комбінації клавіш Ctrl+Alt+Del) і пропонує йому ввести ім'я облікового запису та пароль. При цьому для реалізації такого введення winlogon звертається до спеціальної DLL графічної ідентифікації та аутентифікації (GINA). Стандартне вікно введення даних користувача реалізоване у msgina.dll, програміст може *встановити свою версію цієї динамічної бібліотеки*, що реалізує альтернативний метод аутентифікації (наприклад, на основі смарт-карт або біометричних даних).

Дані від користувача передаються процесу winlogon, і аутентифікація із використанням бази даних SAM (за яку відповідає LSASS) відбувається відповідно до протоколу «виклик-відповідь».

Якщо аутентифікація пройшла успішно, створюють об'єкт, що унікальним чином визначає цього користувача в усіх його подальших діях. Цей об'єкт, який називають **маркером доступу** (access token), відіграє ключову роль у підсистемі захисту: він визначає, до яких системних ресурсів мають доступ потоки, створені цим користувачем.

Після успішної аутентифікації користувача LSASS створює процес і приєднує до нього маркер доступу користувача. Цей процес передають підсистемі Win32, що запускає в його адресному просторі застосування, визначене у реєстрі як оболонка системи (за замовчуванням ним є стандартна оболонка explorer.exe). Застосування формує візуальне відображення параметрів робочого столу для користувача.

Із кожним користувачем і групою у Windows XP пов'язують унікальний **ідентифікатор безпеки** (Security Identifier, SID). Це цілочислове значення, що складається із заголовка і випадкової частини. Система безпеки звертається до користувачів і груп тільки за їхнім SID.

Win32 API надає ряд функцій для роботи із SID. Для отримання SID користувача або групи за іменем використовують функцію LookupAccountName():

```
BOOL LookupAccountName(LPCTSTR sname, LPCTSTR user-name,
PSIO sid, LPDWORD sldsize, LPTSTR dname, LPDWORD dsize,
PSID_NAME_USE sidtype);
```

де: username – ім'я користувача;

sid – покажчик на буфер пам'яті, у який записується отриманий SID, відображений спеціальною структурою S.ID (цей буфер звичайно розміщують у динамічній ділянці пам'яті);

sldsize – розмір буфера (якщо його недостатньо для розміщення SID, у дану змінну записується необхідний розмір, а буфер sid не заповнюється);

dname – покажчик на буфер пам'яті, у який записується ім'я домена, де зареєстрований користувач (для локального користувача – ім'я комп'ютера), dsize – розмір цього буфера;

sidtype – покажчик на змінну, що задає тип SID (вона може набувати значень SidTypeUser – SID користувача, SidTypeGroup – SID групи). Наведемо приклад отримання SID користувача за його ім'ям:

```
DWORD sldsize = 1024, domsize = 1024;
char username[] = "bekas", domname[1024];
SID_NAME_USE sidtype = SidTypeUser;
PSID sid = (PSID)HeapAlloc(GetProcessHeap(), 0, sldsize);
```

```
LookupAccountName(NULL, username, sid, &sidsize, domname,
    &domsize, &sidtype);
// ... робота з sid
FreeSid(sid);
```

Для перетворення структури SID у рядковий формат, придатний для відображення, використовують функцію `ConvertSidToStringSid()`:

```
#include <sddl.h>
char *ssid = NULL;
// PSID sid; LookupAccountName(... sid, ...);
ConvertSidToStringSid(sid, &ssid);
// пам'ять для ssid буде виділено автоматично
printf("%s\n", ssid);
// S-1-5-21-1844237615-1682526488-1202660629-1002
LocalFree(ssid);
```

Маркер доступу користувача пов'язують з усіма процесами, які він створює. Маркер є «посвідченням особи» процесу, коли той намагається використати який-небудь системний ресурс, і містить таку інформацію:

- SID користувача, із правами якого виконується процес;
- список груп, до яких належить цей користувач;
- список привілеїв, якими володіє користувач;
- список контролю доступу за замовчуванням, який визначає первісні права доступу для об'єктів, створюваних процесами цього користувача.

У разі спроби процесу відкрити дескриптор об'єкта диспетчер об'єктів звертається до SRM. SRM отримує маркер доступу, пов'язаний із процесом, та використовує його SID і список груп, щоб визначити, чи має процес право доступу до об'єкта. Фактично маркер доступу відповідає ідентифікатору `euId` для UNIX.

Для отримання маркера доступу процесу використовують функцію `OpenProcessToken()`:

```
BOOL OpenProcessToken(HANDLE ph, DWORD access, PHANDLE ptoken);
```

де: `ph` – дескриптор процесу;

`access` – тип доступу до маркера (`TOKEN_QUERY` – читання інформації);

`ptoken` – покажчик на змінну, в яку записується дескриптор маркера доступу.

Ось приклад отримання маркера доступу поточного процесу:

```
HANDLE mytoken;
```

```
OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &mytoken);
```

Інформація, що зберігається у маркері доступу, може бути отримана за допомогою функції `GetTokenInformation()`:

```
BOOL GetTokenInformation (HANDLE token, TOKEN_INFORMATION_CLASS
    tclass, LPVOID tibuf, DWORD tsize, PDWORD realsize);
```

де: `token` – дескриптор маркера доступу;

`tclass` – значення, що визначає тип отримуваної інформації (`TokenUser` – інформація про обліковий запис користувача, `TokenGroups` – список груп);

`tibuf` – буфер для розміщення інформації (якщо `tclass` дорівнює `TokenUser`, цей буфер містить структуру типу `TOKENUSER` із полем `User`, що відображає структуру із полем `Sid`, яке містить SID цього користувача);

`tsize` – розмір буфера;

`realsize` – покажчик на змінну, що містить розмір буфера, необхідного для

розміщення інформації (якщо `tisize` недостатньо, функція поверне `FALSE` і запише в цю змінну потрібний розмір).

Наведемо приклад отримання `SID` з маркера доступу:

```
GetTokenInformation(mytoken, TokenUser, buf, sizeof(buf), &size);
ConvertSidToStringSid(((PTOKEN_USER)buf)->User.Sid, &ssid);
printf("%s\n", ssid);
```

Окремі потоки процесу можуть виконуватися із правами користувачів, які відрізняються від прав творця цього процесу. Такий підхід називають **запозиченням прав** (*impersonation*), за своїм призначенням він відповідає системному виклику `setuid()` в UNIX-системах. Як і `setuid()`, запозичення прав найчастіше використовують у серверах, які виконує користувач із адміністративними привілеями. У цьому разі потоки, що обслуговують запити користувачів, можуть виконуватися із меншими правами.

Потік, для якого задано запозичення прав, отримує окремий маркер доступу, який називають **маркером режиму запозичення прав** (*impersonation token*). Подальшу перевірку прав доступу виконують із використанням цього маркера.

Для отримання маркера доступу довільного користувача необхідно здійснити програмний вхід у систему для цього користувача. Такий вхід реалізує функція `LogonUser()`:

```
BOOL LogonUser( LPTSTR username, LPTSTR domain, LPTSTR
passwd, DWORD logon_type, DWORD logon_provider, PHANDLE ptoken);
```

де: `username` – ім'я користувача, `domain` – домен або комп'ютер, у якому зареєстрований користувач (для локальної системи задають `"."`),

`passwd` – пароль користувача;

`logon_type` – тип входу у систему (для звичайного користувача потрібно задавати `LOGON32_LOGON_INTERACTIVE`);

`logon_provider` – алгоритм аутентифікації, який використовують для входу (`LOGON32_PROVIDER_DEFAULT` – стандартний алгоритм);

`ptoken` – покажчик на змінну, у яку поміщається маркер доступу цього користувача. Наведемо приклад коду, що здійснює програмний вхід у систему:

```
HANDLE itoken;
LogonUser("bekas", ".", "bekas_pwd", LOGON32_LOGON_INTERACTIVE,
LOGON32_PROVIDER_DEFAULT, &itoken);
```

Реалізацію запозичення прав у користувача, що присутній у системі, здійснюють за допомогою функції `ImpersonateLoggedOnUser()`:

```
ImpersonateLoggedOnUser(itoken);
// ... потік виконується з правами користувача, що володіє itoken
```

Маркер доступу потоку може бути отриманий за допомогою функції `OpenThreadToken()`, аналогічної до `OpenProcessToken()`:

```
OpenThreadToken(GetCurrentThread(), TOKEN_QUERY, TRUE, &ttoken);
// ... GetTokenInformation(ttoken. ...) і т. д.
```

Для повернення до виконання із колишніми правами використовують функцію `RevertToSelf()`:

```
RevertToSelf();
```

Під час створення будь-якого об'єкта Windows XP, який може бути використаний більш як одним процесом (включаючи файли, поіменовані канали, синхронізаційні об'єкти тощо), йому присвоюють **дескриптор захисту** (*security descriptor*).

До найважливіших елементів дескриптора захисту належать:

- `SID` власника об'єкта (власник завжди може змінювати атрибути безпеки об'єкта,

навіть якщо в нього немає прав на доступ до його даних);

- список контролю доступу (ACL), що визначає права доступу до об'єкта. Кожний елемент списку контролю доступу (ACE) містить такі елементи:
- тип ACE (виділяють, зокрема, дозволяючі і забороняючі ACE);
- ідентифікатор безпеки (SID);
- набір прав доступу (читання, записування, повний контроль тощо).

Сума прав доступу, наданих окремими ACE, формує загальний набір прав доступу, наданих ACL.

Розглянемо спрощений приклад ACL для файлового об'єкта (рис. 14.2).

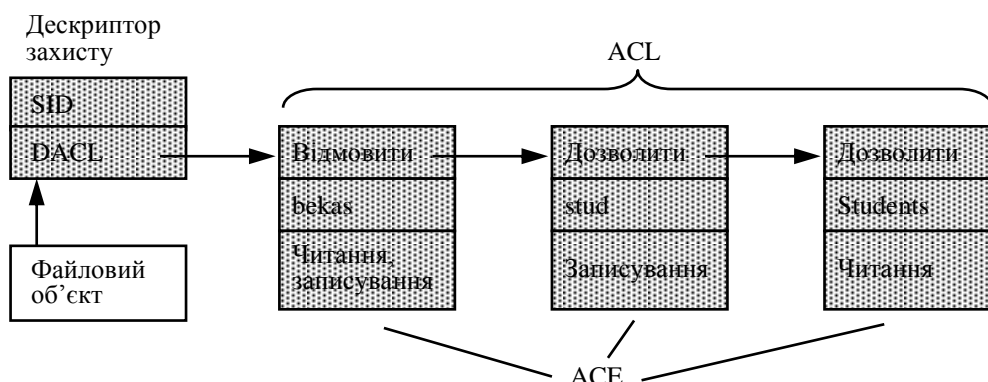


Рис. 14.2. Список контролю доступу

Якщо файловий об'єкт володіє таким ACL, користувач може читати з відповідного файла, коли:

- під час пошуку в ACL файлового об'єкта знайдено ACE, що містить SID цього користувача або однієї із груп, куди він входить; 4 цей ACE дозволяє доступ;
- у ньому є право на читання даних.

Щоб визначити, який ACL має призначатися новому об'єкту, система захисту застосовує три правила в такому порядку.

1. Якщо ACL явно задано під час створення об'єкта, то система захисту присвоює його об'єкту.
2. Якщо ACL не задано, і в об'єкта є ім'я, система захисту шукає ACL каталогу об'єктів, де збережеться ім'я цього об'єкта (для файла таким каталогом буде каталог файлової системи). Деякі ACE каталогу можуть бути позначені як «успадковувані». Це означає, що вони присвоюються новим об'єктам, створеним у цьому каталозі. Якщо такі успадковувані ACE є, із них складають ACL, б що призначають новому об'єкту.
3. Якщо жоден із перших двох випадків не стався, об'єкту присвоюють ACL за замовчуванням із маркера доступу процесу, що робив виклик.

Коли потік намагається відкрити дескриптор об'єкта, диспетчер об'єктів і підсистема захисту зіставляють інформацію із маркера доступу відповідного процесу (що визначає власника процесу) і дескриптора захисту об'єкта (що визначає користувачів, які можуть мати доступ до об'єкта), щоб встановити, чи можна надати потоку запитаний дескриптор.

Подивимося, що відбувається, коли потік, запущений користувачем stud (що входить у групу Students), відкриває файл для читання, викликавши функцію

```
CreateFile("myfile.txt", GENERIC_READ, ...);
```

У цьому разі підсистема безпеки переглядає список контролю доступу для файла myfile.txt, починаючи з першого ACE. Якщо під час перегляду в одному з ACE цього

списку виявляють SID потоку, що робив виклик (узятий із його маркера доступу, в нашому випадку це буде SID для stud), або його групи, перегляд зупиняють і перевіряють, чи дозволяє цей ACE доступ для читання. Коли так, пошук зупиняють, і CreateFile() повертає дескриптор файла. Якщо SID користувача або групи не знайдено (або знайдено в ACE із заборною доступом), CreateFile() поверне помилку.

Елементи, що забороняють доступ, поміщають у початок ACL. Інакше виникла б можлива ситуація, коли дозволяючий ACE з SID-групи, куди входить користувач, перебував перед забороняючим ACE з SID власне цього користувача, внаслідок чого той отримував би доступ до об'єкта, незважаючи на явну заборону.

Якщо ACL для дескриптора безпеки заданий, але не містить жодного елемента, ніхто із користувачів системи не має прав на роботу із цим об'єктом. Якщо ж ACL не заданий (дорівнює NULL), це означає, що будь-який користувач може виконувати будь-які дії з об'єктом. Це становить загрозу безпеці системи, оскільки за такої ситуації зловмисник може змінити права на цей об'єкт, наприклад закрити його для читання від усіх, вивівши цим із ладу застосування, що його використовують.

Система прав доступу в ОС лінії Windows XP досить складна для розуміння і використання. Далі розглянемо простий *приклад задавання прав доступу для читання і записування під час створення файла*.

На першому етапі необхідно отримати доступ до SID, які заносяться у список контролю доступу. Для цього можна використати функцію LookupAccountName().

Далі потрібно розмістити у пам'яті та проініціалізувати список контролю доступу. Для цього треба розрахувати обсяг пам'яті, необхідний для його зберігання, і виділити пам'ять у динамічній ділянці. Цей обсяг отримують додаванням розмірів усіх ACE (що включають розміри ACE-структури і SID) і розміру ACL. Розмір SID повертає функція GetLengthSid():

```
// PSID bekas_sid;
LookupAccountName("bekas", ... bekas_sid, ...);
// два дозволяючих ACE - для читання і записування
DWORD acl_size = 2 * (GetLengthSid(bekas_sid) +
    sizeof(ACCESS_ALLOWED_ACE)) + sizeof(ACL);
PACL pacl = (PACL)HeapAlloc(GetProcessHeap(), 0, acl_size);
```

Після виділення пам'яті її треба проініціалізувати за допомогою функції InitializeAcl(). Першим параметром вона приймає покажчик на виділену пам'ять, другим – її обсяг у байтах

```
InitializeAcl(pacl, acl_size, ACL_REVISION);
```

Кожний елемент контролю доступу потрібно додати в ACL. Для створення і додавання в ACL дозволяючих елементів використовують функцію AccessAllowedAce(), заборонних – AccessDeniedAce():

```
BOOL AddAccessAllowedAce(PACL pacl, DWORD rev, DWORD amask, PSID pSid);
```

де: pacl – покажчик на ACL;

amask – маска прапорців прав доступу (GENERIC_READ, GENERIC_WRITE тощо, повні права задають як GENERIC_ALL);

psid – покажчик на SID, для якого цей елемент задає права.

Ось приклад використання цих двох функцій:

```
// доступ для читання
AddAccessAllowedAce(pacl, ACL_REVISION, GENERIC_READ, bekas_sid);
// доступ для записування
```

```
AddAccessAllowedAce(pacl, ACL_REVISION, GENERIC_WRITE, bekas_sid);
```

На цьому етапі потрібний ACL повністю сформовано. Тепер необхідно сформувати дескриптор безпеки. Для цього слід розмістити у пам'яті структуру типу SECURITY_DESCRIPTOR і проініціалізувати її за допомогою функції InitializeSecurityDescriptor():

```
SECURITY_DESCRIPTOR sd;
```

```
InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION);
```

Далі в цю структуру додають ACL за допомогою функції SetSecurityDescriptorDacl():

```
BOOL SetSecurityDescriptorDacl(PSECURITY_DESCRIPTOR psd,
```

```
    BOOL dacl_present, PACL pdacl, BOOL dacl_from_default);
```

де: psd – покажчик на дескриптор безпеки, для якого задають ACL;

dacl_present – TRUE, якщо ACL задають, FALSE - якщо очищують;

pdacl – покажчик на ACL.

Наведемо код, який додає ACL у дескриптор безпеки.

```
SetSecurityDescriptorDacl(&sd, TRUE, pacl, FALSE);
```

Після створення дескриптора безпеки його потрібно передати у функцію створення файлу. Як уже було видно, у цю функцію (і в багато інших) як параметр передають покажчик на структуру SECURITY_ATTRIBUTES. Покажчик на дескриптор безпеки задано як поле IpSecurityDescriptor цієї структури.

```
SECURITY_ATTRIBUTES sa = { 0 };
```

```
sa.nLength = sizeof(sa) ;
```

```
sa.IpSecurityDescriptor = &sd;
```

```
HANDLE hf = CreateFile("f:\\test.txt", GENERIC_READ | GENERIC_WRITE,  
    0, &sa, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

Після завершення роботи потрібно вивільнити пам'ять, виділену для структур ACL і SID, а також закрити дескриптор файлу:

```
HeapFree(GetProcessHeap(), 0, pacl);
```

```
FreeSid(bekas_sid);
```

```
CloseHandle(hf);
```

14.6. Аудит

Підсистеми аудиту сучасних ОС містять такі основні компоненти.

- Засоби визначення *політики аудиту* (audit policy). Визначивши її, адміністратор системи може задати перелік подій, які його цікавлять. Для зручності визначення політики повідомлення поділяються на рівні залежно від важливості.
- Засоби генерації повідомлень аудиту. Такі повідомлення можуть бути згенеровані різними компонентами режиму ядра і режиму користувача. Для цього ОС має надавати відповідну функціональність і системні виклики.
- Засоби збереження інформації в журналі аудиту, їх зазвичай реалізують централізовано в рамках окремого комп'ютера або мережі, для чого запускають спеціальний фоновий процес, якому передають усю інформацію про повідомлення аудиту.

Треба зазначити, що зловмисник може змінити інформацію в журналі аудиту, щоб приховати сліди свого перебування у системі. Для запобігання таким змінам у системах із високим ступенем захисту журнал аудиту можуть зберігати на віддаленому комп'ютері або на альтернативному носії (наприклад, відразу виводити на принтер або записувати на CD-ROM). Крім того, можна передбачити різні візуальні та звукові сигнали тривоги, що

привертають увагу адміністратора.

Для організації аудиту в UNIX-подібних системах використовують системний журнал (`system log`, `syslog`). За реєстрацію інформації від застосувань у ньому відповідає фоновий процес `syslogd`. Звичайно інформацію записують у файли, що перебувають у каталозі `/var/log` (керують процесом аудиту зміною конфігураційного файла `/etc/syslog.conf`). Усі системні фонові процеси і засоби забезпечення безпеки (наприклад, `login`) зберігають інформацію про свою роботу в цьому журналі. Адміністратори системи мають регулярно переглядати відповідні файли в пошуках можливих порушень політики безпеки.

Прикладні програми також можуть зберігати інформацію про свою діяльність у журналі за допомогою відповідного програмного інтерфейсу.

Зв'язок між процесом користувача і `syslogd` здійснюють через спеціальний сокет. Перед початком використання журналу цей сокет можна відкрити за допомогою системного виклику `openlog()`:

```
#include <syslog.h>
void openlog(char *ident, int options, int facility);
```

де: `ident` – рядок символів, що зберігатиметься в кожному записі журналу (звичайно це назва застосування);

`options` – маска прапорців, що визначають додаткові параметри використання журналу (`LOG_PID` – кожен запис журналу міститиме ідентифікатор процесу, `LOG_CONS` – повідомлення виводяться на консоль у разі неможливості виведення в журнал).

Наведемо код, що відкриває сокет журналу:

```
openlog("myserver", LOG_PID | LOG_CONS, 0);
```

Використання `openlog()` не є обов'язковим (якщо його не виконати, сокет журналу відкривається під час першої спроби записування в нього), але тільки так задають рядок, за яким можна буде розрізняти повідомлення різних процесів.

Записи в журнал здійснюють за допомогою функції `syslog()`:

```
syslog(int level, char *format, ...);
```

де: `level` – рівень серйозності повідомлення (`LOG_INFO` – інформаційне повідомлення, `LOG_WARNING` – попередження, `LOG_ERR` – повідомлення про помилку);

`format` – формат виведення повідомлення (взаємодія цього параметра і наступних аналогічна до `printf()`, за винятком того, що можливий додатковий модифікатор `%m`, який викликає відображення рядка повідомлення про помилку, аналогічного до `perror()`).

Ось як здійснюється реєстрація записів у журналі:

```
syslog(LOG_INFO, "процес запущено, euid=%d", geteuid());
syslog(LOG_WARNING, "неможливо відкрити файл %s: %m", fname);
```

Повідомлення різного рівня серйозності можуть зберігатися в різних файлах. Наприклад, у багатьох Linux-системах задавання рівня `LOG_ERR` призводить до записування повідомлення у файл `/var/log/syslog`, а повідомлення нижчих рівнів – у файл `/var/log/messages`.

Наведемо приклад записів у журналі у разі використання прапорця `LOG_PID`

```
Oct  4 12:44:17 myhost myserver[2074]: процес запущено, euid=0
Oct 14 18:29:31 myhost myserver[2074]: неможливо відкрити файл a.txt;
No such file or directory
```

Після закінчення роботи із журналом його можна закрити (відповідний виклик закриває сокет, виділений для зв'язку із процесом `syslogd`):

```
closelog();
```

Повідомлення аудиту Windows XP можуть бути згенеровані виконавчою підсистемою під час перевірки прав доступу, при цьому за передавання таких повідомлень у режим користувача для реєстрації відповідає SRM. Крім того, повідомлення можуть надходити від компонентів режиму користувача, зокрема від прикладних процесів (для цього у програмах необхідно використовувати відповідні функції Win32 API). Інформацію про такі повідомлення зберігають у **журналі подій** (event log). Для роботи із ним процес повинен мати необхідні привілеї.

Необхідність аудиту конкретної події визначає *локальна політика безпеки* (local security policy). Цю політику підтримує LSASS у межах загальної політики безпеки і передає SRM під час ініціалізації системи. LSASS отримує повідомлення аудиту від SRM або компонентів режиму користувача і відсилає їх окремому процесові журналу подій, що зберігає записи аудиту в журналі.

Розглянемо особливості **використання журналу подій у застосуваннях**. Насамперед необхідно зазначити, що процедура виведення повідомлень у журнал відокремлена від процедури задавання тексту повідомлень. Фактично, під час виведення повідомлення зазначають лише його код, а всю текстову інформацію, що відповідає цьому коду, задають в окремих файлах повідомлень журналу. Спочатку ознайомимося із процесом створення файлів повідомлень, а потім перейдемо до організації виведення повідомлень у журнал.

Вихідні файли повідомлень – це текстові файли спеціального формату, які мають розширення .mc. Структура такого файла проста – він складається з пар ім'я_параметра = значення.

Журнал подій дає змогу задавати повідомлення кількома мовами. При цьому вибір варіанта повідомлення, що відображатиметься під час перегляду журналу за допомогою вікна перегляду подій (Event Viewer), залежить від мовних налаштувань ОС. Для визначення допустимих мов на початку файла повідомлень необхідно задати параметр LanguageNames зі значенням у вигляді списку специфікацій мов, розділених пробілами (у форматі ім'я_мови = код_мови: ім'я_файла):

```
LanguageNames=(English=0x409:msg_en Ukrainian=0x422:msg_ua)
```

Блок інформації для окремого повідомлення має такий вигляд:

```
MessageId=0x1
```

```
SymbolicName=MSG_MYMSG
```

```
language=English
```

```
Application started (executable file: %1)
```

```
.
```

```
Language=Ukrainian
```

```
Програму запущено (виконується файл: %1)
```

```
.
```

Параметр MessageId задає код повідомлення, SymbolicName визначає символічну константу, яка буде доступна в застосуванні як позначення цього повідомлення. Параметр Language починає блок тексту повідомлення, що відповідає заданій мові (ім'я мови треба задати раніше за допомогою параметра LanguageNames). Завершенням такого блоку є рядок із єдиним символом ".". У тексті повідомлення можливі специфікації підстановки %1, %2 тощо, замість них підставляються параметри, задані під час виведення повідомлення.

Після створення mc-файл обробляє компілятор повідомлень (mc.exe).

```
mc -A msgfile.mc
```

Параметр -A означає, що вихідні дані, як і вхідні, будуть у форматі ASCII. У

результаті буде згенеровано такі файли:

- заголовний файл `msgfile.h` із оголошеннями констант, заданих параметрами `SymbolicName`;
- набір файлів із визначеннями текстів для різних мов (наприклад, під час компіляції `msgfile.mc` будуть згенеровані `msg_en.bin` та `msg_ua.bin`);
- файл ресурсів `msgfile.rc`, що містить посилання на файли із визначеннями текстів.
- Після компіляції `mc`-файла потрібно створити двійковий файл повідомлень. Звичайно його компонують у DLL за два етапи:
- текстовий файл ресурсів компілюють у двійкове подання за допомогою компілятора ресурсів


```
rc -r -fo msgfile.res msgfile.rc
```
- на основі двійкового подання ресурсів компонують динамічну бібліотеку


```
link -dll -noentry -out:msgfile.dll msgfile.res
```

Файл повідомлень повинен **реєструватися у системному реєстрі**. Для цього необхідно задати ключ:

```
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
  Application\ім'я джерела повідомлень
```

і в ньому – рядкове значення `EventMessageFile`, яке повинно містити повний шлях до двійкового файла повідомлень. Ім'я джерела повідомлень з'являтиметься у відповідному стовпчику вікна перегляду повідомлень.

Після реєстрації повідомлення, пов'язані із джерелом даних, будуть відображені із використанням зазначеного файла повідомлень.

Після підготовки і реєстрації у системі двійкового файла повідомлень застосування може працювати із журналом подій.

Для **виведення повідомлень у журнал** використовують їхні коди, у ролі яких задають константи з отриманого раніше заголовного файла. Цей файл потрібно підключати до застосувань, що використовують журнал

```
#include "msgfile.h"
```

Перед виведенням повідомлень у журнал необхідно зареєструвати у системі джерело повідомлень, після завершення виведення – скасувати цю реєстрацію. Для цього використовують функції `RegisterEventSource()` і `DeregisterEventSource()`:

```
// myapp повинен збігатися з ім'ям джерела, заданим у реєстрі
HANDLE hlog = RegisterEventSource(NULL, "myapp");
// ... робота з журналом через дескриптор hlog
DeregisterEventSource(hlog);
```

Для виведення повідомлення в журнал використовують функцію `ReportEvent()`:

```
BOOL ReportEvent(HANDLE hlog, WORD etype, WORD ecategory, DWORD ecode,
  PSID userid, WORD parnum, DWORD dsize, LPCTSTR *params, LPVOID data);
```

де: `hlog` – дескриптор журналу;

`etype` – тип повідомлення (`EVENTLOG_ERROR_TYPE` – повідомлення про помилку, `EVENTLOG_WARNING_TYPE` – попередження, `EVENTLOG_INFORMATION_TYPE` – інформаційне повідомлення);

`ecode` – код повідомлення (відповідна константа із заголовного файла);

`parnum` – кількість параметрів повідомлення (елементів масиву `params`);

`params` – масив параметрів повідомлення (під час виведення повідомлення ці параметри будуть послідовно підставлені замість специфікацій підстановки).

Ця функція у разі помилки повертає значення FALSE:

```
const char *messages[] = { argv[0] }: // ім'я виконуваного файла
// HANDLE hlog = RegisterEventSource(...);
ReportEvent(hlog, EVENTLOG_INFORMATION_TYPE, 0,
MSG_MYMSG, NULL, 1, 0, messages, NULL);
```

Унаслідок виконання цього фрагмента коду в журналі подій для джерела myapp у системі із встановленими українськими локальними налаштуваннями відобразиться таке повідомлення:

Програму запущено (виконується файл: c:\path\myapp.exe)

Для використання функцій роботи із журналом під час компонування застосування необхідно зазначати бібліотечний файл advapi32.lib.

14.7. Локальна безпека даних

Механізми керування доступом до файлів не можуть запобігти несанкціонованому доступу до інформації у разі фізичного викрадення жорсткого диска. Зробивши це, **зловмисник** може підключити диск до комп'ютера із сумісною версією операційної системи, у якій він є привілейованим користувачем. Після реєстрації із правами такого користувача можна отримати доступ до всіх файлів на викраденому диску незалежно від того, які для них задані списки контролю доступу. Така проблема характерна для переносних комп'ютерів (ноутбуків), оскільки вони частіше потрапляють у чужі руки.

Щоб запобігти такому розвитку подій, необхідно організувати конфіденційне зберігання найціннішої інформації. Найчастіше це реалізують за допомогою шифрування даних на файлової системі.

Таке *шифрування звичайно здійснюють на рівні драйвера файлової системи*, який перехоплює спроби доступу до файла і шифрує та дешифрує його «на льоту» у разі, коли користувач надав необхідні дані (наприклад, ключ) для виконання цих операцій.

Підтримка шифрувальної файлової системи CFS у Linux реалізована в режимі користувача і дає змогу шифрувати дані на будь-якій наявній файлової системі ціною певної втрати продуктивності. Як алгоритм шифрування використовують потрібний DES. Шифрування може бути застосоване для окремих каталогів і файлів.

Підтримка системи TCFS реалізована в режимі ядра, що дає змогу досягти вищої продуктивності і ступеня захисту. Встановлення підтримки цієї файлової системи, однак, складніше (потрібні внесення змін у код ядра Linux і його перекомпіляція). Ця файлова система реалізує концепцію динамічних модулів шифрування, надаючи користувачу можливість вибору алгоритму і режиму шифрування, які будуть використані для конкретної файлової системи, окремого каталогу або файла.

Засоби підтримки **шифрування файлів у ОС лінії Windows XP** описані під загальною назвою *шифрувальної файлової системи* (Encrypting File System, EFS). Для цього використовують драйвер файлової системи, розташований над драйвером NTFS.

Реалізація EFS – це гібридна криптосистема із кількома рівнями шифрування.

- Безпосередньо для шифрування даних файла використовують симетричний алгоритм (посилений аналог DES, можливе використання інших алгоритмів). Ключ для нього (*ключ шифрування файла* – File Encryption Key, FEK) генерують випадково під час кожної спроби зашифрувати файл.
- Для шифрування FEK використовують алгоритм із відкритим ключем (RSA). Для кожного користувача генерують пару RSA-ключів, при цьому FEK шифрують відкритим ключем цієї пари. Результат шифрування FEK зберігають у заголовку

файла. Крім того, передбачена можливість дешифрування файла довіреною особою (*агентом відновлення, recovery agent*) у разі втрати ключа користувачем. Для цього результат шифрування FEK відкритими ключами довірених агентів також зберігають у заголовку файла.

- Відкритий ключ користувача зберігають у вигляді сертифіката у *сховищі сертифікатів* (certificate store), розташованому в домашньому каталозі користувача на локальному комп'ютері. Крім того, у цьому сховищі містяться сертифікати всіх агентів відновлення.
- Для шифрування закритого ключа користувача використовують симетричний алгоритм RC4 із ключем, який система генерує випадково і періодично оновлює. Цей ключ називають *майстер-ключем* (master key). Результат шифрування закритого ключа майстер-ключем зберігають на файловій системі в домашньому каталозі користувача.
- Для шифрування майстер-ключа теж використовують симетричний алгоритм RC4, але із більшою довжиною ключа. Його генерують на основі застосування односторонньої хеш-функції SHA-1 до даних облікового запису користувача (його SID і паролю). Результат шифрування майстер-ключа цим ключем також зберігають на файловій системі.

Для шифрування файла або каталогу використовують функцію `EncryptFile()`, а для дешифрування – `DecryptFile()`:

```
EncryptFile("myfile.txt");
DecryptFile("myfile.txt", 0);
```

Для перевірки того, чи файл зашифрований, використовують функцію `FileEncryptionStatus()`:

```
BOOL FileEncryptionStatus(LPCTSTR fname, LPDWORD status);
```

де: `fname` – ім'я файла або каталогу;

`status` – покажчик на змінну, у яку заноситься інформація про підтримку шифрування для файла (`FILE_ENCRYPTABLE` – файл може бути зашифрований, `FILE_IS_ENCRYPTED` – файл зашифрований; інші значення показують, що шифрування не підтримується).

Ця функція повертає `FALSE`, якщо під час перевірки виникла помилка.

```
DWORD status;
if (FileEncryptionStatus("myfile.txt", &status)) {
    if (status == FILE_IS_ENCRYPTED)
        printf ("Файл зашифрований\n");
}
```

Для деяких каталогів має сенс заборонити шифрування. Для цього необхідно помістити в такий каталог файл `desktop.ini` з інформацією:

```
[Encryption]
Disable=1
```

14.8. Мережна безпека даних

Розглянемо засоби забезпечення мережної безпеки на різних рівнях мережної архітектури TCP/IP.

Протоколи стека TCP/IP самі не мають достатнього рівня захисту. Фактично, основною їхньою метою була реалізація передавання даних між хостами і застосуваннями. Передбачалося, що для підтримки таких можливостей, як аутентифікація сторін або шифрування даних під час їхнього передавання через канал, ці протоколи потрібно розширити.

Різні *рівні шифрування каналів зв'язку* в рамках стека протоколів TCP/IP наведено нижче.

1. На каналному рівні *шифрують канали зв'язку* (link-by-link encryption). Для цього пакети автоматично шифрують під час передавання через незахищений фізичний канал і дешифрують після отримання. Ці дії звичайно виконує спеціалізоване апаратне забезпечення, у сучасних умовах таке шифрування обмежене модемними або виділеними лініями.
2. На мережному рівні виконують *наскрізне шифрування* (end-to-end encryption), при цьому пакети мережного рівня (наприклад, IP-дейтаграми) шифрують на хості-джерелі, залишають зашифрованими впродовж усього маршруту і автоматично дешифрують хостом-одержувачем. Передавання таких пакетів мережею не змінюється, тому що їхні заголовки залишаються незашифрованими. Прикладом реалізації шифрування на мережному рівні є архітектура протоколів IPSec.
3. На межі між транспортним і прикладним рівнями теж можна шифрувати дані. Наприклад, застосування може бути модифіковане таким чином, що у разі реалізації прикладного протоколу в ньому використовуватиметься не інтерфейс сокетів, а інтерфейс альтернативної бібліотеки, яка реалізує шифрування даних. Такий підхід застосовують у протоколі SSL/TLS.

Розглянемо особливості реалізації двох підходів до шифрування каналів зв'язку:

- наскрізного шифрування на мережному;
- шифрування на транспортному рівнях.

Стандартним підходом для реалізації *захисту інформації на мережному рівні* стека протоколів TCP/IP є архітектура IPSec. Це набір протоколів, призначених для забезпечення наскрізного захисту пакетів, які передають між двома хостами. Розглянемо мережну взаємодію у разі використання IPSec.

1. Хост Аліси генерує IP-дейтаграми для передавання мережею Бобові. Ці дейтаграми порівнюють із фільтрами IPSec, при цьому перевіряють, чи потрібно забезпечувати їхній захист. Фільтри IPSec задають у рамках політики безпеки IPSec, визначеної для хосту Аліси.
2. Якщо дейтаграма відповідає умові фільтра, хост Аліси починає взаємодіяти з хостом Боба із використанням протоколу *обміну ключами Інтернету* (Internet Key Exchange, IKE). При цьому відбувається взаємна аутентифікація сторін (на підставі сертифікатів або пароля), узгодження протоколу захисту пакетів і обмін ключами для шифрування інформації відповідно до протоколу роботи гібридної криптосистеми.
3. Програмне забезпечення хоста Аліси перетворює пакети відповідно до погодженого протоколу захисту пакетів і відсилає їх хосту Боба. У рамках IPSec є два протоколи захисту пакетів:

- *заголовок аутентифікації* (Authentication Header, AH), при використанні якого пакети супроводжують одностороннім хешем для забезпечення цілісності, але не шифрують;
- *інкапсулюючого захищеного навантаження* (Encapsulating Security Payload, ESP), який забезпечує наскрізне шифрування пакетів.

Перетворені пакети – це звичайні IP-дейтаграми, їхнє пересилання мережею відбувається стандартним способом. Додаткові дані, необхідні для протоколів AH і ESP, інкапсулюють усередині дейтаграм.

4. Програмне забезпечення комп'ютера Боба перевіряє пакети на цілісність і дешифрує їхній вміст (для протоколу ESP). Далі IP-пакети передають звичайним способом на верхні рівні реалізації стека протоколів (транспортний і прикладний).

Оскільки протоколи IPSec визначені на мережному рівні, то у разі переходу на IPSec не потрібно змінювати наявне програмне забезпечення – усі дані, передані за допомогою TCP або UDP, будуть автоматично захищені.

Найвідомішим протоколом *захисту інформації на транспортному рівні* був *протокол захищених сокетів* (Secure Socket Layer, **SSL**). Остання його версія (SSL 3.0) із невеликими змінами була прийнята як стандартний *протокол безпеки транспортного рівня* (Transport Layer Security, TLS 1.0); цей протокол називатимемо *SSL/TLS*.

Його розташовують між протоколом транспортного рівня (TCP) і протоколами прикладного рівня (наприклад, HTTP), а реалізацію зазвичай постачають у вигляді бібліотеки користувача, інтерфейс якої можна використати у застосуваннях замість інтерфейсу сокетів. Найвідомішою реалізацією бібліотеки підтримки цього протоколу є OpenSSL.

Застосування може реалізовувати протокол прикладного рівня поверх інтерфейсу SSL/TLS. Є кілька реалізацій стандартних протоколів прикладного рівня, які базуються на SSL/TLS, серед них реалізація протоколу HTTP (HTTPS), підтримувана більшістю сучасних веб-серверів і веб-браузерів.

Розглянемо послідовність кроків у разі використання SSL/TLS на прикладі HTTPS. Цей протокол ґрунтується на протоколі роботи гібридної криптосистеми.

1. Клієнт (веб-браузер) зв'язується із сервером із використанням протоколу HTTPS (наприклад, запросивши веб-документ, URL якого починається із https://). При цьому встановлюється TCP-з'єднання із портом 443 (стандартний порт HTTPS).
2. Клієнт і сервер погоджують алгоритми шифрування, які використовуватимуться під час обміну даними.
3. Сервер відсилає клієнтові свій відкритий ключ у складі сертифіката.
4. Клієнт верифікує сертифікат сервера за допомогою відкритого ключа довіреної сторони (центру сертифікації), що видала цей сертифікат. Сучасні веб-браузери постачають із ключами деяких центрів сертифікації, можна також встановлювати додаткові ключі. Якщо сертифікат вірний, аутентифікацію сервера вважають успішною.
5. Клієнт генерує сесійний ключ, шифрує його відкритим ключем сервера, шифрує HTTP-запит сесійним ключем і відсилає всю цю інформацію серверу.
6. Сервер розшифровує сесійний ключ своїм закритим ключем, HTTP-запит – сесійним ключем, формує HTTP-відповідь, шифрує її сесійним ключем, доповнює одностороннім хешем і відсилає клієнтові.
7. Клієнт дешифрує HTTP-відповідь сесійним ключем і перевіряє цілісність (для чого обчислює її односторонній хеш і порівнює із хешем, отриманим від сервера). Якщо цілісність дотримана, документ відобразиться.

Ще одним важливим протоколом *безпеки транспортного рівня* є протокол *безпечного командного інтерпретатора* (Secure Shell, **SSH**). Він багато в чому подібний до SSL/TLS, але використовується інакше. Найчастіше він застосовується для реалізації захищеного віддаленого доступу до системи.

Після встановлення з'єднання між SSH-клієнтом і SSH-сервером (коли завершена аутентифікація сторін і обмін ключами) SSH-сервер на віддаленому комп'ютері запускає копію командного інтерпретатора, яка використовує псевдотермінал. SSH-клієнт може взаємодіяти із цим інтерпретатором, емулюючи термінал.

Взаємодія користувача із системою фактично відбувається аналогічно до протоколу telnet, але при цьому весь канал зв'язку шифрують. Сьогодні у багатьох UNIX-системах використання SSH є обов'язковим (протокол telnet блокують із міркувань безпеки).

14.9. Атаки і боротьба з ними

Ознайомимося із деякими підходами, які можна використати для атаки на систему безпеки ОС. Обмежимося атаками переповнення буфера і найпростішими прикладами відмови в обслуговуванні.

Розповсюдженим типом атак на програмний код у сучасних ОС є *атаки переповнення буфера* (buffer overflow attacks). Усі вони використовують некоректний програмний код (переважно мовою C), що не перевіряє довжину буфера, у який записують зовнішні дані, отримані від користувача. Ось приклад такого коду:

```
#include <stdio.h>
void f() {
char buf[128];
gets(buf);
// небезпечне одержання рядка даних зі стандартного вводу }
```

Функція `gets()`, що входить у стандартну бібліотеку мови C, вводить рядок символів довільної довжини зі стандартного вводу і розміщує їх у буфері `buf`. При цьому сама функція не перевіряє, скільки символів насправді було введено і чи достатньо для них місця в буфері. У ситуації, коли користувач ввів більше ніж 128 символів, програма запише дані у пам'ять, розташовану за `buf`.

Для того щоб зрозуміти, у чому тут полягає небезпека, розглянемо організацію пам'яті, у якій виконують застосування (рис. 14.3).

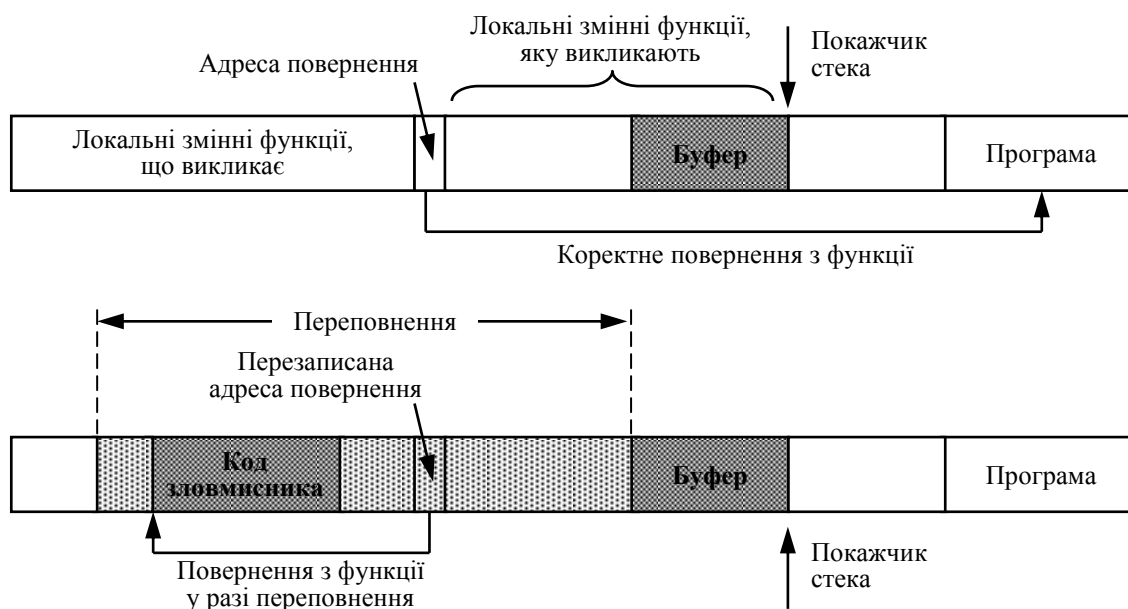


Рис. 14.3. Переповнення буфера

Як бачимо, адреса повернення функції і локальні змінні (зокрема і буфер `buf`) розміщені у програмному стеку. Коли зловмисник введе значно більше символів, ніж може бути розміщено у `buf`, вони переповнять буфер і будуть записані поверх адреси повернення функції. У результаті після повернення із функції відбудеться перехід за адресою, взятою із введеного зловмисником рядка. При цьому вміст рядка може бути ретельно підібраний так, щоб цей його фрагмент містив адресу коду, який бажає виконати зловмисник. Згаданий код може перебувати в іншій частині цього самого рядка і, наприклад, запускати командний інтерпретатор. Якщо застосування виконувалося із правами суперкористувача, запущений інтерпретатор дасть зловмисникові повний контроль над системою.

Для захисту від таких атак насамперед необхідно підвищувати якість розробки програмного забезпечення. Необхідно повністю відмовитися від використання функцій, які не перевіряють обсягу введених даних, замінивши їх варіантами, що роблять таку перевірку. Наприклад, замість функції `gets()` потрібно використати `fgets()`:

```
char buf[128];
fgets(buf, sizeof(buf), stdin);
// введення не більше 128 символів
```

Крім `gets()`, подібні проблеми виникають у разі використання таких функцій, як `strcpy()`, `strcat()` і `sprintf()`. Замість них потрібно використовувати відповідно `strncpy()`, `strncat()` і `snprintf()`.

Захист від переповнення буфера на рівні ОС може полягати в цілковитій забороні виконання коду, що перебуває у програмному стеку. Для Linux є виправлення до коду ядра, що реалізують це обмеження.

Найпростіший **приклад атаки** відмови в обслуговуванні для UNIX-систем – так звана «fork-бомба»:

```
void main() {
for (; ;) fork(); }
```

Очевидно, що процес, завантажений у пам'ять внаслідок запуску такої «бомби», почне негайно створювати свої власні копії, те саме продовжать робити ці копії і т. д. У старих версіях UNIX це могло швидко привести систему до непрацездатного стану, і навіть сьогодні запуск такого застосування у системі із малим обсягом ресурсів здатний значно понизити її продуктивність.

Для боротьби із такими атаками необхідно встановлювати ліміти на ресурси. Зокрема, потрібно, щоб у системі було встановлено ліміт на максимальну кількість процесів, які можуть бути запущені під обліковим записом користувача. За замовчуванням ця кількість дорівнює 256, змінити її можна за допомогою системного виклику `setrlimit()`:

```
#include <sys/resource.h>
#include <unistd.h>
struct rlimit plimit;
plimit.rlim_max = 100;
setrlimit(RLIMIT_NPROC, &plimit);
```

Для ліквідації наслідків такої атаки не можна намагатися негайно завершити всі процеси-бомби, виконавши, наприклад, команду вилучення всіх процесів із заданим ім'ям:

```
# killall -KILL bomb
```

Річ у тому, що після запуску «бомби» у системі швидко створюється стільки

процесів, що ліміт на їхню кількість виявиться вичерпаним. Після цього всі процеси-бомби перестають створювати нащадків і залишаються у нескінченному циклі. Як тільки після виконання `killall` завершиться один із таких процесів (а вони всі не можуть завершитись одночасно), загальна кількість процесів стане менша за ліміт. У результаті якийсь інший процес набору відразу отримує можливість виконати `fork()` і встигає це зробити до свого завершення. Фактично після знищення поточного набору процесів його місце негайно займає новий.

Правильним підходом буде спочатку призупинити всі процеси-бомби, а потім послати їм сигнал завершення:

```
# killall -STOP bomb
# killall -KILL bomb
```

Системний виклик `setrlimit()` може також бути використаний для встановлення інших квот на ресурси, зокрема обмежень на кількість відкритих файлів (першим параметром потрібно задати `RLIMIT_NOFILE`), на частку процесорного часу, яку використовує процес (`RLIMIT_CPU`), на розмір процесу у пам'яті (`RLIMIT_AS`).

Другим важливим засобом обмеження споживання ресурсів є **квоти дискового простору**.

Дискову квоту – встановлюване адміністратором обмеження на використання різних ресурсів файлової системи – задають для кожного користувача. Зазвичай до таких ресурсів належать кількість виділених кластерів або кількість створюваних файлів. Операційна система гарантує, що користувачі не зможуть перевищити виділені їм квоти.

Квоти звичайно реалізують так. На диску організовують спеціальний файл (файл квот), що містить таблицю квот, куди заносять інформацію про квоти різних користувачів і те, які поточні обсяги ресурсів ними витрачені. Якщо користувач відкрив хоча б один файл, відповідний цьому користувачу елемент таблиці квот зчитують у пам'ять.

Файловий дескриптор, відкритий процесом, містить покажчик на елемент таблиці квот, що відповідає користувачу, який створив процес. У разі зміни розміру якогось із відкритих файлів інформацію про нову сумарну витрату ресурсу зберігають в елементі таблиці у пам'яті. Після того як користувач закриє всі свої файли, елемент таблиці квот зберігають на диску у файлі квот.

В елементі таблиці квот міститься така інформація: гнучкий і жорсткий ліміт на ресурс; поточний стан витрати ресурсу; максимально допустима і поточна кількість попереджень про перевищення гнучкого ліміту.

Спроба перевищити жорсткий ліміт завжди спричиняє помилку. Гнучкий ліміт може бути перевищений користувачем, але він при цьому отримує попередження (для кожного користувача визначено скінчений запас таких попереджень). Якщо користувач витратить усі попередження про перевищення гнучкого ліміту, його більше не допускають у систему без дозволу адміністратора.

Дискові квоти підтримують у Linux і в лінії Windows XP (для файлової системи NTFS).

Одним із ефективних способів запобігання або ослаблення дії різних атак в UNIX-системах є **обмеження видимості файлової системи для застосувань**. У такому разі кореневим каталогом ("/") для процесу стає не справжній кореневий каталог файлової системи, а один із його підкаталогів. Поза даним каталогом файлова система для цього процесу буде недоступною. Таку зміну відображення файлової системи здійснюють за допомогою системного виклику `chroot()`, у даному випадку кажуть про застосування зі зміненим кореневим каталогом або `chroot`-застосування.

Застосування зі зміненням кореневим каталогом не має доступу до більшості каталогів системи, тому заподіяна шкода від зловмисника, який отримав контроль над системою за допомогою такого застосування (наприклад, за допомогою атаки переповнення буфера), буде значно обмежена.

Виклик `chroot()` може бути виконаний тільки процесом із правами суперкористувача. Наведемо приклад його використання для того щоб зробити кореневим каталогом поточного процесу `/home/user/newroot`:

```
#include <unistd.h>
chroot("/home/user/newroot");
```

Зміну кореневого каталогу часто використовують у різних серверах. Так, деякі сучасні версії стандартних мережних серверів (сервер імен `bind`, поштовий сервер `sendmail`) як один із режимів підтримують роботу за умов зміненого кореневого каталогу.

Розглянемо послідовність завантаження сервера у разі використання `chroot()`.

1. Здійснюють підготовку до обслуговування клієнтів (завантажують необхідні динамічні бібліотеки і конфігураційні файли, відкривають файли журналу).
2. Виконують виклик `chroot()`.
3. Сервер починає очікування з'єднань від клієнтів та їхнє обслуговування. При цьому рекомендовано перейти в режим виконання із правами звичайного користувача за допомогою системного виклику `setuid()`.

Висновки

- Основними завданнями забезпечення безпеки комп'ютерних систем є аутентифікація, керування доступом, аудит, забезпечення конфіденційності, доступності та цілісності даних.
- Сучасні засоби підтримки безпеки ґрунтуються на таких криптографічних технологіях, як алгоритми із секретним і відкритим ключами, гібридні криптосистеми, цифрові підписи і сертифікати.
- Аутентифікація дає змогу впевнитися, що користувач є тим, за кого себе видає, і може бути допущений у систему. Для підтвердження особи користувача звичайно використовують пароль. Сучасні технології аутентифікації дають змогу уникнути передавання пароля каналом зв'язку у незашифрованому вигляді.
- Керування доступом (авторизація) дає змогу визначити дії, які авторизований користувач може виконувати із різними об'єктами у системі. Авторизація реалізована на основі визначення списків контролю доступу, пов'язаних із об'єктами у системі, а також списків можливостей, пов'язаних із окремими користувачами.
- Організація аудиту дає змогу зберігати інформацію про різні події у системі для подальшого аналізу. Інформацію зберігають у спеціальному журналі аудиту (системному журналі, журналі подій), вибір подій для реєстрації в цьому журналі визначає політика аудиту.
- Для забезпечення конфіденційності даних у рамках локальної системи використовують шифрувальні файлові системи або криптографічні API. Мережна безпека даних забезпечена їхнім шифруванням на різних рівнях мережної архітектури. Найчастіше таке шифрування виконують на мережному рівні або між транспортним і прикладним рівнями.

Контрольні запитання та завдання

1. Як можна реалізувати підтримку цілісності повідомлень і конфіденційності передачі даних у системі електронної пошти?

2. Наведено описи трьох систем керування доступом ОС:

а) кожен файл містить список коректних паролів; користувач зобов'язаний пред'явити один із цих паролів, щоб відкрити файл;

б) кожному файлу присвоюють ім'я з 256 випадкових символів, єдиний спосіб одержати доступ до файла – це задати його ім'я;

в) для кожного файла адміністратор системи задає список користувачів, яким заборонено доступ до файла. Яку технологію керування доступом використовують у кожній з цих систем?

3. Поясніть, яким чином введення підтримки списків контролю доступу може розширити функціональність системи, яка базується на можливостях.

4. Що спільного й у чому основні відмінності між роботою з журналом файлової системи і використанням журналу безпеки?

5. Модифікуйте фонові застосування, розроблені під час виконання завдань 9 і 10 з розділу 17, так, щоб виведення діагностичної інформації виконувалося, відповідно, у системний журнал (Linux) і в журнал повідомлень (Windows XP).

6. Припустімо, що в системі зареєстровано 4000 користувачів, і необхідно, щоб 3990 з них могли одержати доступ до одного файла. Як досягти цього в UNIX? Які поліпшення схеми атрибутів безпеки UNIX можна запропонувати з урахуванням можливості виникнення такої ситуації?

7. Розробіть застосування для Windows XP, що створює каталог і встановлює права читання і записування у цей каталог для заданого користувача. Імена каталогу і користувача задають у командному рядку.

8. Як виконати атаку відмови в обслуговуванні з використанням підсистеми керування віртуальною пам'яттю сучасних ОС?

9. Мережні хробаки – це застосування, що автоматично поширюють свої копії доступними мережними з'єднаннями. Яким чином поява в системі такого хробака може призвести до атаки відмови в обслуговуванні, навіть якщо він не містить коду, що зумисно здійснює таку атаку?

10. Користувач victim залишив на якийсь час без догляду консоль Linux з відкритою сесією командного інтерпретатора. Користувач attacker, використовуючи цю консоль, виконав дії, що дають йому змогу надалі в будь-який момент одержати повний контроль над усіма даними victim. Незважаючи на те, що victim знає про дії attacker, перешкодити захопленню контролю над своїми даними він не може. Ні victim, ні attacker не мають права root. Які дії виконав attacker?

11. Яким чином користувач може дозволити іншим користувачам додавати рядки у файл, власником якого він є? Всі інші дії з файлом (вилучення, зміна рядків тощо) мають бути заборонені. Є як мінімум два принципово різних розв'язання цієї задачі. Розробіть їхній програмний код.