МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

В.В. Давидов, С.Г. Семенов, Н.Г. Кучук, С.С. Бульба

СУЧАСНІ ТЕХНОЛОГІЇ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ

Навчально-методичний посібник
для студентів спеціальності 123
«Комп’ютерна інженерія»

Затверджено редакційно-видавничою радою
університету, протокол №1 від 25.02.2021 р.

Харків
НТУ «ХПІ»
2021

УДК 004.05 004.4
С 91

Рецензенти:
Г.А. Кучук, д-р техн. наук, проф. НТУ «ХПІ»
А.А. Коваленко, д-р техн. наук,проф. Харківського національного університету радіоелектроніки

Авторський колектив В.В Давидов, С.Г Семенов, Бульба С.С.

С 91 Сучасні технології безпечного програмування: навч.-метод. посіб. / Давидов В.В., Семенов С.Г., Кучук Н.Г., Бульба С.С. – Харків : НТУ «ХПІ», 2021. – 117 с.

ISBN

Викладено існуючі методи захисту програмного забезпечення, розглянуто сучасні технології безпечного програмування. Матеріал проілюстровано практичними прикладами, до всіх розділів наведено необхідні завдання та приклади.

Для студентів спеціальності 123 «Комп’ютерна інженерія».

Табл. 4. Іл. 16 Бібліогр. 8

УДК 004.05 004.4

ISBN

© Давидов В.В., Семенов С.Г.,
Кучук Н.Г., Бульба С.С, 2021 р.

ВСТУП

Збільшення залежності людського суспільства від мережевих програмних систем супроводжується збільшенням кількості нападів, скоєних на ці системи. Такі напади - спрямовані проти урядів, корпорацій, навчальних закладів та приватних осіб - призводять до втрати і компрометації конфіденційних даних, пошкодженню систем, втрат продуктивності і фінансових втрат. Хоча проблема сьогоднішніх атак в Інтернеті — це не більше, ніж просто невелика незручність, велика кількість більше свідчень того, що злочинці, терористи й інші шкідники шукають і використовують уразливості в програмних системах для досягнення своїх цілей. Сьогодні швидкість відкриття вразливостей програмного забезпечення становить понад 4000 на рік. Це викликано проектуванням і реалізацією програмного забезпечення, які не захищають системи належним чином, і поширеною практикою програмування, коли програмісти недостатньо зосереджуються на ліквідації дефектів реалізації, які призводять до появи уразливості. Спостерігається також стійкий прогрес у витонченості і ефективності атак зловмисників, що швидко розробляють нові шкідливі сценарії, які використовують виявлені в програмах уразливості. Потім ці сценарії запроваджуються для несанкціонованого доступу до інформації, а взаємообмін такими сценаріями сприяє їх широкому розповсюдженню. Ці сценарії в поєднанні з програмами, автоматично скануючими мережу в пошуках уразливих систем, і їх

атаки призводять до вибухоподібного поширення нападів на обчислювальні системи.

Велика кількість виявлених щороку вразливостей призводить до перевантаженості адміністраторів роботою з виправлення існуючих систем. Виправлення для цих програм можуть бути важко застосовними або мають несподівані побічні ефекти. З того моменту, коли розробник випускає оновлення безпеки, і до моменту його установлення на 90-95 % уразливих комп'ютерів можуть пройти місяці або навіть роки. Раніше користувачі Інтернету в значній мірі спиралися на можливості інтернет-спільноти в цілому досить швидко реагувати на атаки на систему безпеки і зводити завдані збитки до мінімуму. Сьогодні, однак, стає ясно, що ми досягли межі ефективності наших реактивних рішень. Незважаючи на всі зусилля окремих організацій з раціоналізації та автоматизації процедур виправлень, кількість вразливостей в комерційних програмних продуктах сьогодні знаходиться на рівні, що не дозволяє ніяким, крім самих забезпечених будь-якими ресурсами, організаціям йти в ногу з новими виправленнями вразливостей. Є дуже мало свідчень поліпшень в області безпеки для більшості програмних продуктів. Багато розробників програмного забезпечення не засвоїли уроки з виявлених вразливостей і не застосовують належні стратегії для пом'якшення наслідків. Про це свідчить той факт, що CERT / CC1 продовжує знаходити ті ж типи вразливостей в останніх версіях продуктів, які були і в попередніх версіях. Взяті разом, ці фактори вказують, що слід очікувати значних економічних втрат від атак і збоїв в обслуговуванні навіть підчас відгуку, меншому, ніж ми можемо реально сподіватися досягти. Агресивний, скоординовану відповідь як і раніше необхідний, але ми повинні удосконалені системи, зламати які буде істотно складніше.

У даному навчально-методичному посібнику описані основні помилки при програмуванні мовами високого рівня при розробленні веб-додатків, які можуть призводити до найбільш поширених, небезпечних і руйнівних вразливостей програмного забезпечення. Проведено як поглиблений аналіз інженерних програмних помилок, які призводять до таких уражень, так і наведено стратегії, що можуть бути ефективно застосовано для зменшення або усунення ризику використання цих вразливостей зловмисниками.

ЗМІСТ

ЛАБОРАТОРНА РОБОТА №1 6
ЛАБОРАТОРНА РОБОТА №2 18
ЛАБОРАТОРНА РОБОТА №3 36
ЛАБОРАТОРНА РОБОТА №4 44
ЛАБОРАТОРНА РОБОТА №5 52
ЛАБОРАТОРНА РОБОТА №6 62
ЛАБОРАТОРНА РОБОТА №7 75
ЛАБОРАТОРНА РОБОТА №8 84
ЛАБОРАТОРНА РОБОТА №9 92
ЛАБОРАТОРНА РОБОТА №10 100
СПИСОК ЛІТЕРАТУРИ 109

ЛАБОРАТОРНА РОБОТА 1

Вступ до стеганографії

Мета роботи: дослідити можливість «приховування» даних у зображеннях.

1.1. Теоретичні відомості

Стеганографія – це спосіб передачі або зберігання інформації з урахуванням збереження в таємниці самого факту такої передачі. На відміну від криптографії, яка приховує вміст таємного повідомлення, стеганографія приховує сам факт його існування. Частіше за все, повідомлення буде виглядати як дещо інше, наприклад, зображення, стаття, список покупок, лист або судоку. Стеганографію зазвичай використовують спільно з методами криптографії, таким чином, доповнюючи її.

Перевага стеганографії над чистою криптографією полягає в тому, що повідомлення не привертають до себе уваги. Повідомлення, факт шифрування яких не прихований, викликають підозру і можуть бути самі по собі бути викриті в тих країнах, де заборонена криптографія. Таким чином, криптографія захищає вміст повідомлення, а стеганографія захищає сам факт його існування.

Розглянемо деякі методи «приховування» даних у зображеннях.

Технологія RAR-JPEG

Зміст цієї технології полягає в приховування архіву з даними всередині графічного файлу формату JPEG. RAR-JPEG отримали найбільшого поширення, хоча операція з'єднання графічного файлу та архіву можлива також для інших графічних форматів (PNG, GIF, навіть BMP, і т. і.) та архівів (ZIP, 7Z, в тому числі і JAR-застосунків на платформі Java), також аудіо- та відеофайлів у форматі Ogg.

За використання JPEG та RAR справа полягає в тому, що у структурі JPEG файлу є маркер що, позначає кінець зображення, і вся інформація розташована після такого маркера, не береться до уваги при читанні файлу. У

той же час в архіві є маркер, який вказує на початок архіву, і вся інформація, яка міститься до цього маркера, просто не помічається архіватором. Наприклад, можна ховати ключі і паролі, фотографії або вести таємне листування. Якщо ви сховаєте таку картинку або фото серед інших зображень на своєму комп'ютері, то ніхто не здогадається, де ви ховаєте свої секрети. Навіть знаючи цей спосіб, досить важко знайти потрібне зображення в масі інших картинок (особливо, якщо їх надто багато).

Єдине, що може видати таку хитрість, це великий розмір картинки, але цей спосіб добре підходить для маленьких файлів.

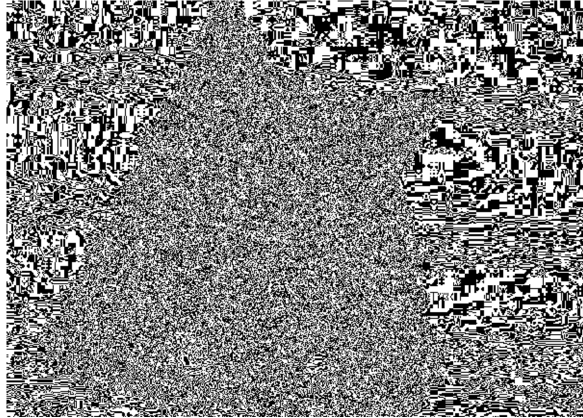
Метод LSB

LSB (Least Significant Bit, найменший значущий біт) — суть цього методу полягає в заміні останніх значущих бітів у контейнері (зображення, аудіо або відеозапису) на біти приховуваного повідомлення. Цей метод є найбільш поширеним в електронній стеганографії. Він базується на обмежених можливостях людських органів почуттів, в силу який люди не здатні розрізняти незначні варіації кольорів. Кожна компонента кодується в класичному варіанті за допомогою 8 біт, тобто може приймати значення від 0 до 255. Саме тут і приховується найменш значущий біт. Важливо зрозуміти, що на один RGB-колір відводиться аж три таких біта.

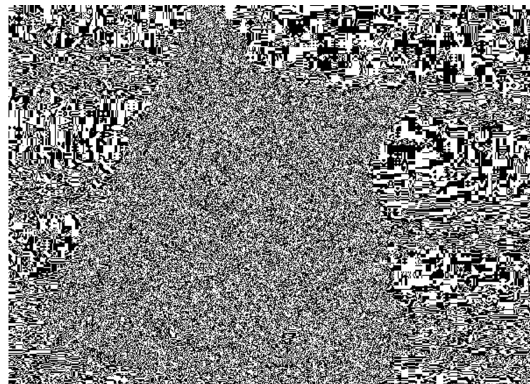
Як приклад для демонстрації роботи методу LSB оберемо картинку у png форматі (рис. 1.1).

Рисунок 1.1 – Початкове зображення

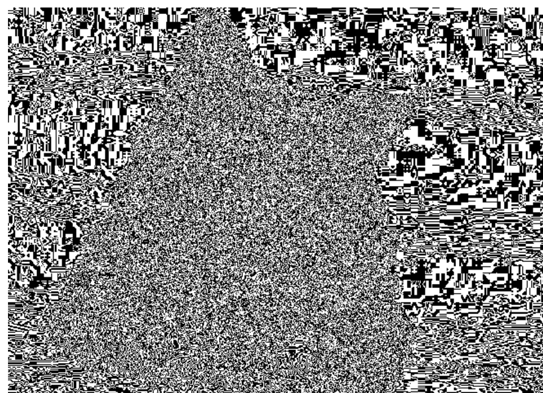
Розіб'ємо її на три канали і в кожному каналі візьмемо найменш значущий біт. Створимо три нових зображення, де кожен піксель позначає LSB. Нуль – піксель білий, одиниця – чорний (рис. 1.2)



а



б



в

Рисунок 1.2 – Зображення, де кожен піксель позначає LSB (а) для червоного, (б) зеленого та (в) синього каналів

Для того, щоб уявити LSB трьох компонент в одному зображенні (рис 1.3), достатньо компоненту в пікселі, де LSB дорівнює одиниці, замінити на 255, і у зворотному випадку замінити на 0.

Рисунок 1.3 – НЗБ трьох компонент

Далі уявимо показане на рис. 1.3 як потік бітів, звідки ми можемо читати і куди ми можемо записувати інформацію. Беремо дані, які ми хочемо вставити в зображення, подаємо їх у вигляді бітів та послідовно записуємо на місце вже існуючих. Для вилучення цих даних прочитаємо LSB як потік бітів та приведемо до потрібного вигляду. Щоб дізнатися, скільки бітів потрібно зчитати, як правило, в початок записують розмір повідомлення.

Потрібно відзначити, що приблизно в 50 % випадків біт, який ми хочемо записати, і біт у зображенні будуть збігатися. У такому випадку змінювати нічого не доведеться.

Нижче наведені зображення, які є незаповненим (рис. 1.4) та заповненим (рис. 1.5) стеганоконтейнерами.

Рисунок 1.4 – Незаповнений стеганоконтейнер

Рисунок 1.5 – Стеганоконтейнер, що заповнено на 95 %

На вигляд ці зображення не відрізняються одне від одного. Розглянемо, чому це відбувається.

Подивимося на два кольори: (0, 0, 0) та (1, 1, 1), тобто на кольори, що відрізняються тільки на LSB в кожній компоненті (рис. 1.6).

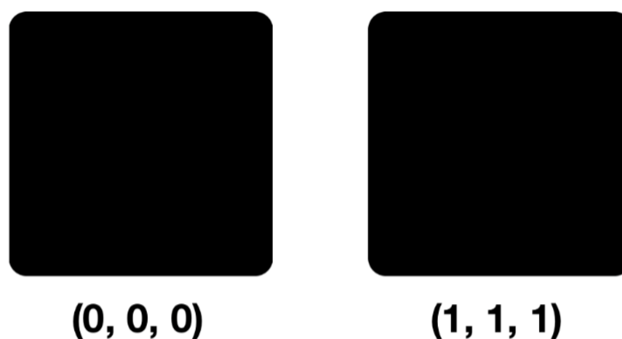


Рисунок 1.6 – Кольори, що відрізняються на LSB

Невеликі відмінності в пікселях при першому погляді помітні не будуть. Справа в тому, що око людини може розрізнити близько 10 мільйонів кольорів, а мозок всього лише близько 150. Модель RGB ж містить 16777216 кольорів.

Метод LSB займає лідуючі позиції серед стеганографічних алгоритмів. Це можна пояснити декількома причинами. По-перше, мультимедіаконтейнери не викликають підозр: можна без проблем надіслати другу свою фотографію або пейзаж. По-друге, молодші біти оцифрованих зображень, звуку або відео можуть мати різний розподіл залежно від застосування параметрів аналого-цифрового перетворення, від додаткової комп'ютерної обробки та від інших факторів. Ця особливість робить метод найменш значущих бітів найбільш захищеним від виявлення вкладення. Нарешті, по-третє, реалізації LSB для більшості стандартів файлів-контейнерів не вимагають значних витрат часу і сил,бо ідея зазначеного методу дуже проста.

Крім таємного пересилання та зберігання інформації, стеганографія має ще одну сферу застосування – це захист авторських прав. Оскільки до зображення, аудіо- або відеофайлу можна додати будь-яке повідомлення таким чином, щоб це не зіпсувало враження від перегляду/прослуховування, і оскільки таке вкладення практично неможливо виявити і вилучити, то це повідомлення можна використовувати як авторський підпис. Подібні «водяні знаки» допоможуть довести, що, наприклад, зроблена вами фотографія була незаконно використана для оформлення якогось відомого Web-сайту.

Стеганоаналіз

Стегоаналіз або **Стеганоаналіз** – розділ стеганографії; наука про виявлення факту передачі прихованої інформації в повідомленні. У деяких випадках під стеганоаналізом розуміють також витяг прихованої інформації з повідомлення, що містить її, і (якщо це необхідно) подальше її дешифрування.

Першою в списку атак на LSB-стеганографії виступає візуальна атака.

Так, у заповненому стеганоконтейнері є специфічний для приховування повідомлення «рисунок» в НЗБ (рис. 1.7). На перший погляд це здається простим шумом, але при більш уважному розгляді проглядається структура. Тут видно, що стегоконтейнер заповнений. Якби ми взяли повідомлення в 30 % від місткості обраної картинки, то отримали трохи інший результат (рис. 1.8).

Рисунок 1.7 – НЗБ повністю заповненого стеганоконтейнера

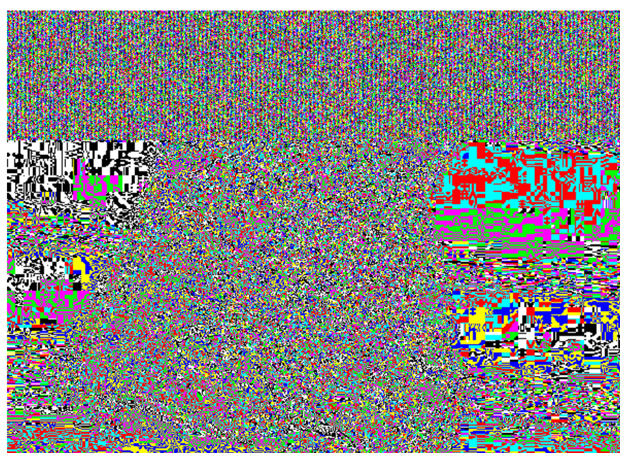


Рисунок 1.8 – НЗБ стеганоконтейнера, заповнененого на 30%

Визначимо, скільки всього інформації можна «сховати» у обраному стеганоконтейнері та скільки її там знаходиться зараз. Розмір обраної як приклад картинки 603x433 пікселів. 30 % від цієї величини дорівнює 78459 пікселям. У кожен піксель поміщається 3 біта інформації. Разом 78459 * 3 = 235377 біт або трохи менше, ніж 30 кілобайт поміщається у 30 %. А в цілу картинку поміститься близько 100 кілобайт.

Атака «Хі-квадрат»

Цей метод був запропонований у 2000 році Андресом Вестфелд і Андреасом Пфітцманом. Атака «Хі-квадрат» ґрунтується на тому припущенні, що ймовірність одночасної появи сусідніх (відмінних один від одного на найменш значущий біт) кольорів (pair of values) в незаповненому стегоконтейнері вкрай мала. Тобто кількість пікселів двох сусідніх кольорів істотно відрізняється для порожнього контейнера.

Нехай h – масив, на i-му місці містить кількість пікселів i-го кольору в досліджуваному зображенні.

Маємо:

1. Виміряна частота появи кольору $i = 2k$:

$$n_k = h[2k], k \epsilon [0, 127];$$

2. Теоретично очікувана частота появи кольору:

$$n_k^* = \frac{h[2k] + h[2k+1]}{2}, k \in [0,127];$$

Хі-квадрат критерій для кількості ступенів свободи *k*-1 розраховується таким чином (*k* – кількість різних кольорів, тобто 256):

$$\chi_{k-1}^2 = \sum_{i=1}^{k} \frac{(n_k - n_k^*)^2}{n_k^*}$$

P – це ймовірність того, що розподіли при цих умовах рівні (ймовірність того, що перед нами заповнений стегоконтейнер). Вона розраховується за допомогою інтегрування функції гладкості:

$$P = 1 - \frac{1}{2^{\frac{k-1}{2}}\Gamma\left(\frac{k-1}{2}\right)} \int_0^{\chi_{k-1}^2} e^{-\frac{x}{2}} x^{\frac{k-1}{2}-1} dx.$$

Найефективніше застосовувати *Xi*-квадрат не до всього зображення, а тільки до його частин, наприклад, до рядків. Якщо порахована ймовірність для рядка більше 0.5, то рядок в оригінальному зображенні закрасимо червоним, якщо менше – то зеленим. Для зображення-прикладу з 30 %, що заповнено на 30 %, картина буде виглядати, як на рис. 1.9.

Рисунок 1.9 – Застосовувати *Xi*-квадрат

RS-аналіз

RS-аналіз (Regular-Singular аналіз) – це атака на підставі відомого заповненого контейнера на систему вбудовування стего в зображення методом LSB. Regular-Singular аналіз був запропонований в 2001 році колективом дослідників з Бінгемтонского університету.

Метод ґрунтується на поділі зображення на пов'язані групи G по n пікселів. Для кожної групи визначається значення функції регулярності або гладкості $f(G)$. Найчастіше функція регулярності – це сума перепадів сусідніх пікселів у групі.

Також вводиться функція фліппінга – функція F така, що $F(F(x)) = x$. При цьому аналізі використовують три функції фліппінга:

- F_1 – інверсія молодшого біта кольору в зображенні;
- F_0 – залишення без змін;
- F_2 – інверсія молодшого біта кольору в зображенні з перенесенням у старший біт.

Усередині групи можна застосовувати різні функції фліппінга для різних пікселів, тому записують маску M – n-мірний вектор у просторі {-1,0,1}, що вказує, яким пікселям в такій групі встановлюється який фліппінг:

$$F(G) = (F_{M(1)}(x_1), \dots, F_{M(n)}(x_n))$$

Всі отримані групи G ділять на три види:

- Регулярні, для яких $F(G)$ збільшує значення гладкості
- Сингулярні, для яких $F(G)$ зменшує значення гладкості
- Невикористані, для яких $F(G)$ не змінює значення гладкості

Далі підраховують кількість R_M регулярних груп, кількість S_M сингулярних груп для маски M і аналогічні величини R_{-M}, S_{-M} для інвертованої маски $-M$. Статистична гіпотеза дослідників, підтверджена дослідженням вибірки з реальних фотографій, полягає в тому, що інвертування маски майже не змінює кількості регулярних і сингулярних груп для порожнього контейнера:

$$R_M \approx R_{-M},\ S_M \approx S_{-M}$$

У той же час дослідники помітили, що внесення випадкових спотворень в таке співвідношення порушує дане співвідношення так, що випадкові спотворення зменшують різницю між R_M і зі збільшенням довжини впроваджуваного повідомлення. На цьому факті будується метод *RS*-аналізу:

1. Будують діаграму: на осі абсцис відкладають частку інвертованих біт, на осі ординат відкладають частки сингулярних і регулярних груп з усіх
2. На діаграмі S_M отримують кілька ліній, припускаючи довжину повідомлення p і частку зміни молодших біт при записі повідомлення 50 %:

2.1 Прямі R_{-M} і S_{-M} будують за двома точками: при незмінному зображенні (тобто в точці з абсцисою $p/2$) і при зображенні з інвертованими молодшими бітами (тобто в точці з абсцисою $1–p/2$)

2.2 Параболи R_M і S_M будують за трьома точками: в точці з абсцисою $p/2$, в точці з абсцисою $1–p/2$ і в точці з абсцисою 50 % (записавши в молодші біти випадкові значення).

3. Прийнявши абсциссу $p/2$ за 0 і абсциссу $1–p/2$ за 1, визначають абсцису x точки перетину кривих R_M та S_M та вважають передбачувану довжину повідомлення:

$$p = \frac{x}{x-1/2}.$$

1.2. Індивідуальне завдання

1. Навести реалізацію технології RAR-JPEG, та продемонструвати її роботу.
2. Виконати укриття даних у зображення за допомогою методу найменш значущих бітів (Less Significant Bits).
3. Виконати аналіз укриття даних за допомогою методу стеганоаналізу «атака Xi-квадрат»

Додаткове завдання (опціональне, на додаткові бали):

Виконати аналіз скриття даних за допомогою RS-методу стеганоаналізу.

Контрольні запитання

1. Що таке стеганографія?
2. Чим відрізняється стеганографія від криптографії?
3. Які є області використання стеганографії?
4. Що таке технологія RAR-JPEG?
5. Для яких форматів можлива операція з'єднання графічного файлу та архіву?
6. Чому є можливим використання технології RAR-JPEG?
7. У чому полягає суть методу LSB?
8. На чому базований метод LSB?
9. Чому людина не бачить різниці між кольорами, що відрізняються тільки на LSB в кожній компоненті?
10. Чому метод LSB вважається одним з найпопулярніших стеганографічних методів?
11. Що таке стеганоаналіз?
12. Як визначити, чи заповнений стеганоконтейнер, за допомогою візуальної атаки?
13. На чому базується метод «атака *Xi*-квадрат»?
14. Як визначити, чи заповнений стеганоконтейнер за допомогою «атаки Xi-квадрат»?
15. У чому полягає зміст *RS*-методу?

ЛАБОРАТОРНА РОБОТА 2

Зберігання паролів

Мета роботи: дослідити і порівняти існуючі механізми зберігання паролів.

2.1. Теоретичні відомості

Хешування паролів є одним із основних міркувань безпеки, які необхідно використати, при розробленні програми, що приймає паролі від користувачів. Без хешування, паролі, що зберігаються в базі вашого застосування, можуть бути вкрадені, наприклад, якщо ваша база даних була скомпрометована, а потім негайно можуть бути застосовані для компрометації не тільки вашої програми, але і акаунтів ваших користувачів на інших сервісах, якщо вони не використовують унікальних паролів.

Застосовуючи хешуючий алгоритм до призначених для користувача паролів перед збереженням їх у своїй базі даних, можна зробити неможливим розгадування оригінального пароля для атакуючого базу даних, в той же час зберігаючи можливість порівняння отриманого хешу з оригінальним паролем.

Важливо зауважити, однак, що хешування паролів захищає їх тільки від компрометації у вашому сховищі, але необов'язково від втручання шкідливого коду у застосунку.

До геш-функцій належать:

- Cyclic redundant check (CRC): CRC16, CRC32
- Message-Digest Algorithm (MD): MD2, MD5, MD6
- Message authentication code (MAC): HMAC, OMAC, KMAC
- Secure hash algorithm (SHA): SHA-1, SHA-256
- BCrypt, SCrypt

MD5

MD5 (Message Digest 5) – це 128-бітний алгоритм хешування, розроблений Рональдом Л. Рівестом в 1991 році. Він призначений для створення «відбитків» або «дайджестів» повідомлень довільної довжини. MD5 прийшов на зміну недосконалому MD4, але з 2011 року відповідно до RFC 6151 він також вважається ненадійним. Незважаючи на це, на сьогоднішній день MD5 все ще широко використовується.

Раніше вважалося, що MD5 дозволяє отримувати відносно надійний ідентифікатор для блоку даних. На даний момент ця геш-функція не рекомендується до використання, оскільки існують способи знаходження колізій з прийнятною обчислювальною складністю.

За допомогою MD5 перевіряли цілісність та автентичність відбитку завантажених файлів – так, деякі програми поставляються разом зі значенням контрольної суми. Наприклад, пакети для інсталяції вільного ПЗ.

MD5 використовувався для хешування паролів. В системі UNIX кожен користувач має свій пароль, та його знає тільки користувач. Для захисту паролів використовується хешування. Передбачалося, що отримати справжній пароль можна тільки повним перебором. При появі UNIX єдиним способом хешування був DES (Data Encryption Standard), але ним могли користуватися тільки жителі США, тому що вихідні коди DES можна було вивозити з країни. Під FreeBSD вирішили цю проблему. Користувачі США могли використовувати бібліотеку DES, а решта користувачів мають метод, дозволений для експорту. Тому в FreeBSD стали використовувати MD5 за замовчуванням. Деякі Linux-системи також використовують MD5 для зберігання паролів.

Багато систем використовують бази даних для автентифікації користувачів і існує кілька способів зберігання паролів:

1. Паролі зберігаються як є. При зломі такої бази всі паролі стануть відомі.

2. Зберігаються тільки хеші паролів. Знайти паролі можна, використовуючи заздалегідь підготовлені таблиці хешів. Такі таблиці складаються з хешів простих або популярних паролів.
3. До кожного пароля додається кілька випадкових символів (їх називають «сіль»), і результат хешується. Отриманий хеш разом з «сіллю» зберігаються у відкритому вигляді. Знайти пароль за допомогою таблиць таким методом не вийде.

Існує кілька надбудов над MD5.

MD5 (HMAC) – Keyed-Hashing for Message Authentication (змішування з ключем для аутентифікації повідомлення) – алгоритм дозволяє хешувати вхідне повідомлення L з деяким ключем K, таке змішування дозволяє аутентифікувати підпис.

MD5 (Base64) – тут отриманий MD5-хеш кодується алгоритмом Base64.

MD5 (Unix) – алгоритм викликає тисячу разів стандартний MD5 для ускладнення процесу. Також відомий як MD5crypt.

Алгоритм MD5

Вхідні дані вирівнюються так, щоб їхній розмір можна було порівняти з 448 за модулем з 512. Спочатку дописують одиничний біт (навіть якщо довжина порівняна з 448), далі необхідна кількість нульових бітів.

Дописування 64-бітного представлення довжини даних з вирівнювання. Якщо довжина перевищує $2^{64} – 1$, то дописують молодші біти.

Ініціалізують 4 змінних розміром по 32 біта:

A = 01 23 45 67;

B = 89 AB CD EF;

C = FE DC BA 98;

D = 76 54 32 10.

Вирівнювані дані розбиваються на блоки по 32 біта, і кожен проходить 4 раунди з 16 операторів. Всі оператори однотипні і мають вигляд: [abcd k s i], визначений як $a = b + ((a + Fun(b, c, d) + X[k] + T < i >) <<< s)$, де X – блок даних, а T[1..64] – 64-елементна таблиця, побудована таким чином: $T[i] = int(4294967296 * |sin(i)|)$, s – циклічний зсув вліво на s біт отриманого 32-бітного аргументу.

- В першому раунді Fun F(X, Y, Z) = XY v (not X)Z
- В другому раунді Fun G(X, Y, Z) = XZ v (not Z)Y.
- В третьому раунді Fun H(X, Y, Z) = X xor Y xor Z.
- В четвертому раунді Fun I(X, Y, Z) = Y xor (X v (not Z)).

Саме обчислення проходить таким чином:

Зберігаються значення *A, B, C і D*, що залишились після операцій з попередніми блоками (або їх початкові значення, якщо блок перший)

AA = A

BB = B

CC = C

DD = D

Раунд 1

/*[abcd k s i] a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */

[ABCD 0 7 1][DABC 1 12 2][CDAB 2 17 3][BCDA 3 22 4]

[ABCD 4 7 5][DABC 5 12 6][CDAB 6 17 7][BCDA 7 22 8]

[ABCD 8 7 9][DABC 9 12 10][CDAB 10 17 11][BCDA 11 22 12]

[ABCD 12 7 13][DABC 13 12 14][CDAB 14 17 15][BCDA 15 22 16]

Раунд 2

/*[abcd k s i] a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */

[ABCD 1 5 17][DABC 6 9 18][CDAB 11 14 19][BCDA 0 20 20]

[ABCD 5 5 21][DABC 10 9 22][CDAB 15 14 23][BCDA 4 20 24]

[ABCD 9 5 25][DABC 14 9 26][CDAB 3 14 27][BCDA 8 20 28]

[ABCD 13 5 29][DABC 2 9 30][CDAB 7 14 31][BCDA 12 20 32]

Раунд 3

/*[abcd k s i] a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */

[ABCD 5 4 33][DABC 8 11 34][CDAB 11 16 35][BCDA 14 23 36]

[ABCD 1 4 37][DABC 4 11 38][CDAB 7 16 39][BCDA 10 23 40]

[ABCD 13 4 41][DABC 0 11 42][CDAB 3 16 43][BCDA 6 23 44]

[ABCD 9 4 45][DABC 12 11 46][CDAB 15 16 47][BCDA 2 23 48]

Раунд 4

/*[abcd k s i] a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */

[ABCD 0 6 49][DABC 7 10 50][CDAB 14 15 51][BCDA 5 21 52]

[ABCD 12 6 53][DABC 3 10 54][CDAB 10 15 55][BCDA 1 21 56]

[ABCD 8 6 57][DABC 15 10 58][CDAB 6 15 59][BCDA 13 21 60]

[ABCD 4 6 61][DABC 11 10 62][CDAB 2 15 63][BCDA 9 21 64]

Виконати такі операції:

$A = AA + A$

$B = BB + B$

$C = CC + C$

$D = DD + D$

Після цього перевірити, чи є ще блоки, якщо є, то повторюють циклічну процедуру обчислення для наступного 32 - бітового блоку.

Після обчислення для всіх блоків даних, отримуємо кінцевий Хеш у регістрах *A B C D*. Якщо вивести слова у зворотному порядку *DCBA*, то отримаємо MD5 хеш.

MD5-Хеш містить 128 біт (16 байт) і зазвичай подається як послідовність з 32 шістнадцяткових цифр.

Наприклад, MD5 ("md5") = 1bc29b36f623ba82aaf6724fd3b16718

Навіть невелика зміна вхідного повідомлення (у нашому випадку на один біт: ASCII символ «5» з кодом 0x35 16 = 00011010 '1' 2 замінюється на символ «4» з кодом 0x34 16 = 00011010 '0' 2) призводить до повної зміни хешу. Така властивість алгоритму називається лавинним ефектом.

Наступні криптографічні бібліотеки підтримують MD5: Botan, Bouncy Castle, cryptlib, Crypto++, Libgcrypt, Nettle, OpenSSL, wolfSSL.

Bcrypt

bcrypt – адаптивна криптографічна функція формування ключа, що використовується для безпечного зберігання паролів. Функція базується на шифрі Blowfish, вперше представлена на USENIX у 1999 році. Для захисту від атак за допомогою райдужних таблиць bcrypt використовує сіль (salt); крім того, функція є адаптивною, час її роботи легко налаштовується і її можна сповільнити, щоб ускладнити атаки перебором.

Шифр Blowfish відрізняється від багатьох алгоритмів обчислювально складною фазою підготовки ключів шифрування. Розробки скористалися цією особливістю, але змінили алгоритм підготовки ключів, отримавши шифр «Eksblowfish» (expensive key schedule Blowfish). Кількість раундів у підготовці ключів має бути ступенем двійки; конкретна ступінь може задаватися при використанні bcrypt.

Спочатку реалізовано функції crypt в OpenBSD. Існують реалізації для Java, Python, Nim, C#, Ruby, Perl, PHP 5.3, Node.js та деяких інших.

Алгоритм bcrypt використовує алгоритм налаштування ключів з «Eksblowfish»:

EksBlowfishSetup(cost, salt, key)

state InitState()

state ExpandKey(state, salt, key)

repeat (2cost)

state ExpandKey(state, 0, key)

state ExpandKey(state, 0, salt)

return state

Функція InitState відповідає оригінальній функції з шифру Blowfish; для заповнення масиву P і S-box використовується дробна частина числа

Функція ExpandKey:

```
ExpandKey(state, salt, key)
   for(n = 1..18)
      Pn   key[32(n-1)..32n-1]   Pn //treat the key as cyclic
   ctext   Encrypt(salt[0..63])
   P1   ctext[0..31]
   P2   ctext[32..63]
   for(n = 2..9)
      ctext   Encrypt(ctext   salt[64(n-1)..64n-1]) //encrypt using the current key schedule and treat the salt as cyclic
      P2n-1)   ctext[0..31]
      P2n   ctext[32..63]
   for(i = 1..4)
      for(n = 0..127)
         ctext   Encrypt(ctext   salt[64(n-1)..64n-1]) //as above
         Si[2n]   ctext[0..31]
         Si[2n+1]   ctext[32..63]
   return state
```

Для обчислення хешу bcrypt обробляє вхідні дані еквівалентно шифруванню 'eksblowfish (посилений_ключ, input)':

```
bcrypt(cost, salt, key, input)
 state   EksBlowfishSetup(cost, salt, key)
 ctext   input
 repeat(64)
 ctext   EncryptECB(state, ctext) // шифрування стандартним Blowfish в режимі ECB
```

return Concatenate(cost, salt, ctext)

В різних ОС (linux, OpenBSD), використовують алгоритм bcrypt в стандартній функції crypt (3), в якості input подається константа «OrpheanBeholderScryDoubt».

bcrypt був розроблений в 1999 році і був захищений від ефективного перебору на апаратних засобах того часу. В даний час одержали широке розповсюдження ПЛІС, в яких bcrypt реалізується ефективніше. У 2009 році був створений алгоритм scrypt, що вимагає для своєї роботи значного обсягу пам'яті (з випадковим доступом), об'єм пам'яті налаштовується.

SHA-1

Secure Hash Algorithm 1 – алгоритм криптографічного хешування. Для вхідного повідомлення довільної довжини алгоритм генерує 160-бітове хеш-значення, відоме також дайджестом повідомлення.

SHA-1 реалізує хеш-функцію, побудовану на ідеї функції стиснення. Входом функції стиснення є блок повідомлення довжиною 512 біт і вихід попереднього блоку повідомлення. Виходом є значення всіх хеш-блоків до цього моменту. Іншими словами, хеш блоку $h_i = f(M_i, h_{i-1})$. Хеш-значення всього повідомлення є виходом останнього блоку.

Вхідне повідомлення розбивається на блоки по 512 біт у кожному. Останній блок доповнюється до довжини, кратної 512 біт. Спочатку додається 1, а потім нулі, щоб довжина блоку стала рівною (512-64=448) біт. В останні 64 біта записується довжина вихідного повідомлення у бітах (в big-endian форматі). Якщо останній блок має довжину понад 448, але менше 512 біт, то додаток виконується в такий спосіб: спочатку додається 1, потім нулі аж до кінця 512-бітного блоку; після цього створюється ще один 512-бітний блок, який заповнюється аж до 448 біта нулями, після чого в 64 біта, що залишилися, записується довжина вихідного повідомлення в бітах (в big-endian форматі). Доповнення останнього блоку здійснюється завжди, навіть якщо повідомлення вже має потрібну довжину.

Ініціалізуються п'ять 32-бітових змінних:

A = a = 0x67452301

B = b = 0xEFCDAB89

C = c = 0x98BADCFE

D = d = 0x10325476

E = e = 0xC3D2E1F0

Визначаються чотири нелінійні операції і чотири константи (табл. 2.1).

Таблиця 2.1 – Визначення нелінійних операцій та констант

$F_t(m,l,k) = (m \wedge l) \vee (\neg m \wedge k)$	K_t = 0x5A827999	0≤t≤19
$F_t(m,l,k) = m \oplus l \oplus k$	K_t = 0x6ED9EBA1	20≤t≤39
$F_t(m,l,k) = (m \wedge l) \vee (m \wedge k) \vee (l \wedge k)$	K_t = 0x8F1BBCDC	40≤t≤59
$F_t(m,l,k) = m \oplus l \oplus k$	K_t = 0xCA62C1D6	60≤t≤79

Головний цикл ітеративно обробляє кожен 512-бітний блок. Ітерація складається з чотирьох етапів по двадцять операцій у кожному. Блок повідомлення перетворюється з 16 32-бітових слів *Mi* у 80 32-бітових слів *Wj* за таким правилом:

при 0≤t≤15 = (-3 -8 -14 -16) << 1

при 16≤t≤79

тут << — це циклічний зсув вліво

для від 0 до 79

temp = (a << 5) + (b,c,d) + e +

e = d

d = c

$c = b << 30$

$b = a$

$a = temp$

Після цього *a, b, c, d, e* додаються до *A, B, C, D, E* відповідно. Починається наступна ітерація. Підсумковим значенням буде об'єднання п'яти 32-бітних слів в одне 160-бітове хеш-значення.

Порівняння SHA-1 з MD5

MD5 і SHA-1 є, по суті, поліпшеними версіями MD4.

Схожість:

1. Чотири етапи.
2. Кожна дія додається до раніше отриманого результату.
3. Розмір блоку обробки становить 512 біт.
4. Обидва алгоритми виконують складання за модулем 232, вони розраховані на 32 — бітну архітектуру.

Відмінності:

1. У SHA-1 на четвертому етапі використовується та ж функція *f*, що і на другому етапі.
2. В MD5 у кожній дії використовується унікальна адитивна константа. У SHA-1 константи використовуються повторно для кожної із чотирьох груп.
3. У SHA-1 додана п'ята змінна.
4. SHA-1 використовує циклічний код виправлення помилок.
5. В MD5 чотири зсуви, що використовуються на кожному етапі, відрізняються від значень, які використовуються на попередніх етапах. В SHA на кожному етапі використовується постійне значення зсуву.
6. В MD5 чотири різних елементарних логічних функції, в SHA-1 – три.

7. В MD5 довжина дайджесту становить 128 біт, в SHA-1 – 160 біт.
8. SHA-1 містить більше раундів (80 замість 64) і виконується на 160-бітному буфері у порівнянні із 128-бітовим буфером MD5. Таким чином, SHA-1 повинен виконуватися приблизно на 25 % повільніше, ніж MD5 на тій же апаратурі.

Сіль

Сіль (Salt, також модифікатор) – рядок даних, який передається геш-функції разом з паролем.

Сіль використовується переважно для захисту від перебору за словником і атак з використанням райдужних таблиць, а також приховування однакових паролів. Однак, сіль не може захистити від повного перебору кожного окремого пароля.

Сіль використовується для захисту паролів при їх зберіганні. Раніше паролі зберігались у відкритому вигляді на серверах у файлах або базах даних, що не забезпечувало їх захист у разі несанкціонованого доступу до серверу або у випадках викрадення файлу. Для протидії таким загрозам з часом створювали додаткові методи захисту паролів при їх зберігання. Сіль – один з таких методів.

Сіль генерується випадковим чином для кожного пароля. Сіль та пароль об'єднують і ей рядок перетворюється за допомогою криптографічної хеш-функції в геш, який і зберігається разом із сіллю. Це дозволяє перевірити пароль без його збереження.

Наприклад, хешуєте і зберігаєте свої паролі в MD5. Якщо ваша база буде вкрадена – зловмисник досить просто відновить більшість вихідних паролів, використовуючи заздалегідь підготовлені райдужні таблиці. Якщо ж ми «посолимо» пароль, тобто з'єднаємо рядок з 10—20 випадковими символами з паролем і вже від цього рядка знайдемо MD5, – стандартні таблиці не будуть працювати, тому що вони не розраховані на пошук такого довгого рядка.

Проблеми, пов'язані з сіллю і надійністю паролів:

При несанкціонованому доступі до бази даних або вдалій SQL-ін'єкції зловмисник отримає дані доступу одного або декількох користувачів. Якби паролі зберігалися в первісному вигляді, зловмисник міг би спробувати використовувати їх для доступу до інших ресурсів (таким чином відбувається захист користувача сайту від злому профілів в інших системах – у нього буде час на зміну паролів, поки зловмисник зайнятий підбором.)

Існує безліч функцій для створення гешів як складних, так і простих, до того ж кожен може написати свою реалізацію. Однак, все зводиться до того, як швидко буде отримано доступ до використання такої ж хешуючої функції і генерування райдужної таблиці.

Одна з найважливіших місій солі – зробити різними хеші паролів в тому випадку, якщо двоє вказали однаковий пароль, тим самим ускладнивши перебір. Це ж актуально за умови, що одній людині дозволено мати кілька профілів.

Райдужна таблиця

Райдужна таблиця (англ. Rainbow table) – спеціальний варіант таблиць пошуку (англ. Lookup table) для обернення криптографічних хеш-функцій, що використовує механізм розумного компромісу між часом пошуку по таблиці і займаною пам'яттю (англ. Time-memory tradeoff). Райдужні таблиці використовуються для розкриття паролів, перетворених за допомогою складнозворотної хеш-функції, а також для атак на симетричні шифри на основі відомого відкритого тексту. Використання функції формування ключа із застосуванням солі робить цю атаку нездійсненною.

Райдужні таблиці є розвитком більш раннього і простого алгоритму, запропонованого Мартіном Хеллманом.

Комп'ютерні системи, які використовують паролі для аутентифікації, повинні якимось чином визначати правильність введеного пароля. Найпростішим способом вирішення даної проблеми є зберігання списку всіх допустимих паролів для кожного користувача. Мінусом даного методу є те, що в разі несанкціонованого доступу до списку зловмисник дізнається всі призначені для

користувача паролі. Більш поширений підхід полягає в зберіганні значень криптографічної хеш-функції від парольної фрази. Однак більшість хешів швидко обчислюється, тому зловмисник, який отримав доступ до хешів, може швидко перевірити список можливих паролів на валідність. Щоб уникнути цього, потрібно використовувати більш довгі паролі, тим самим збільшуючи список паролів, які повинен перевірити зловмисник. Для простих паролів, що не містять солі, зломщик може заздалегідь підрахувати значення гешів для всіх поширених і коротких паролів і зберегти їх в таблиці. Тепер можна швидко знайти збіг в заздалегідь отриманій таблиці. Але чим довше пароль, тим більше таблиця, і тим більше пам'яті необхідно для її зберігання. Альтернативним варіантом є зберігання тільки перших елементів ланцюжків хешів. Це потребує більше обчислень для пошуку пароля, але значно зменшить кількість необхідної пам'яті. А райдужні таблиці є поліпшеним варіантом даного методу, які дозволяють уникнути колізій.

Нехай у нас є геш-функція H з довжиною хеш n і кінцева множина паролів P. Наша мета – створити структуру даних, яка для будь-якого значення хешу h може знайти такий елемент p з P, що $H(p)=h$, або визначити, що такого елемента не існує. Найпростіший спосіб зробити це – обчислити H(p) для всіх p з P, але для зберігання такої таблиці потрібно $\Theta(|P|n)$ пам'яті, що дуже багато.

Ланцюжки хешів – метод для зменшення цієї вимоги до обсягу пам'яті. Головна ідея – визначення функції редукції R, яка зіставляє значенням геша значення P. Зауважимо, що R не є зверненням хеш-функції. Починаючи з вихідного пароля і поперемінно застосовуючи до кожного отриманого значення H і R, ми отримаємо ланцюжок перемежованих паролів і хешів. Наприклад, для набору паролів довжиною в 6 символів і хеш-функції, що видає 32-бітні значення, ланцюжок може виглядати так:

$$aaaaaa \underset{H}{\rightarrow} 281DAF40 \underset{R}{\rightarrow} sgfnyd \underset{H}{\rightarrow} 920ECF10 \underset{R}{\rightarrow} kiebgt$$

До функції редукції висувається єдина вимога: повертати значення з того ж алфавіту, що і паролі.

Для генерації таблиці ми вибираємо випадкову множину початкових паролів з P, обчислюємо ланцюжок деякої фіксованої довжини k для кожного пароля і зберігаємо тільки перший і останній паролі з кожного ланцюжка.

Для кожного хешу h, значення якого ми хочемо обернути (знайти відповідний йому пароль), обчислюємо послідовність *R(…R(H(R(h)))…).* Якщо деяке з проміжних значень зійдеться з яким-небудь кінцем будь-якого ланцюжка, ми беремо початок цього ланцюжка і відновлюємо його повністю. З високою ймовірністю повний ланцюжок буде містити значення хешу h, а передуючий йому пароль буде шуканим.

Для прикладу, зазначеного вище, якщо у нас є хеш 920ECF10, він породить таку послідовність:

$$920ECF10 \underset{R}{\rightarrow} kiebgt$$

Оскільки *kiebgt* є кінцем ланцюжка з нашої таблиці, ми беремо відповідний початковий пароль *aaaaaa* і обчислюємо ланцюжок, поки не знайдемо хеш 920ECF10:

$$aaaaaa \underset{H}{\rightarrow} 281DAF40 \underset{R}{\rightarrow} sgfnyd \underset{H}{\rightarrow} 920ECF10$$

Таким чином, шуканий пароль — *sgfnyd.*

Варто зауважити, що відновлений ланцюжок не завжди містить шукане значення хешу h. Таке можливо при виникненні колізії функції H або **R**. Наприклад, нехай дано геш FB107E70, який на певному етапі породжує пароль kiebgt:

$$FB107E70 \underset{R}{\rightarrow} bvtdll \underset{H}{\rightarrow} 0EE80890 \underset{R}{\rightarrow} kiebgt$$

Але FB107E70 не з'явиться в ланцюжку, породженому паролем аааааа. Це називається помилковим спрацюванням. У цьому випадку, ми ігноруємо збіг і продовжуємо визначати послідовність, породжену *h*. Якщо згенерована послідовність досягає довжини *k* без хороших збігів, це означає, що шуканий пароль ніколи не зустрічався в підготовлених ланцюжках.

Вміст таблиці, не залежить від значення досліджуваного хешу, вона обчислюється заздалегідь і використовується лише для швидкого пошуку. Збільшення довжини ланцюжка зменшує розмір таблиці, але збільшує час пошуку потрібного елемента в ланцюжку.

Райдужні таблиці є розвитком ідеї таблиці хеш-ланцюгів. Вони ефективно вирішують проблему колізій шляхом введення послідовності функцій редукції $R_1, R_2, \ldots, R_k$. Функції редукції застосовуються по черзі, перемежаючись з функцією гешування: $H, R_1, H, R_2, \ldots, H, R_k$.

При такому підході два ланцюжки можуть злитися тільки за умови збігу значень на тій самій ітерації. Отже, достатньо перевіряти на колізії тільки кінцеві значення ланцюжків, що не вимагає додаткової пам'яті.

На кінцевому етапі складання таблиці можна знайти всі злиті ланцюжки, залишити з них тільки один і згенерувати нові, щоб заповнити таблицю необхідною кількістю різних ланцюжків. Отримані ланцюжки не є повністю вільними від колізій, тим не менш, вони не зіллються повністю.

Використання послідовностей функцій редукції змінює спосіб пошуку по таблиці. Оскільки хеш може бути знайдений в будь-якому місці ланцюжка, необхідно згенерувати k різних ланцюжків:

– перший ланцюжок будується на припущенні, що шуканий хеш зустрінеться на останній позиції в табличному ланцюжку, тому складається з єдиного значення *Rk(h)*;

– другий ланцюжок будується на припущенні, що шуканий хеш зустрінеться на передостанній позиції в табличному ланцюжку, тому виглядає так: *Rk(H(Rk-1(h)))*;

– аналогічно, нарощуючи довжину ланцюжка і застосовуючи функції редукції з меншими номерами, отримуємо інші ланцюжки. Останній ланцюжок буде мати довжину *k* і містити всі функції редукції: *Rk(H(Rk-1(H(…H(R1(h))…))))*

Також зміниться і визначення помилкового спрацювання: якщо ми неправильно «вгадаємо» позицію шуканого хешу, це буде відкидати тільки один з *k* згенерованих ланцюжків; при цьому все ще може залишатися можливість знайти вірний хеш для даного табличного ланцюжка, але на іншій позиції.

Хоча райдужні таблиці вимагають відстеження більшої кількості ланцюжків, вони мають велику щільність кількості паролів на один табличний запис. На відміну від таблиці хеш-ланцюжків, застосування декількох функцій редукції зменшує кількість потенційних колізій у таблиці, що дозволяє нарощувати їх без небезпеки отримати велику кількість злиття ланцюжків.

При генерації таблиць важливо знайти найкраще співвідношення взаємопов'язаних параметрів:

– ймовірність знаходження пароля за отриманим и таблицями;
– час генерації таблиць;
– час підбору пароля по таблицях;
– займане місце.

Названі вище параметри залежать від налаштувань, заданих при генерації таблиць:

– допустимий набір символів;
– довжина пароля;
– довжина ланцюжка;
– кількість таблиць.

При цьому час генерації залежить майже виключно від бажаної ймовірності підбору, використовуваного набору символів і довжини пароля. Займане таблицями місце залежить від бажаної швидкості 1 підбору пароля по готових таблицях.

Хоча застосування райдужних таблиць полегшує використання повного перебору (тобто методу грубої сили — bruteforce) для підбору паролів, в деяких випадках необхідні для їх генерації/використання обчислювальні потужності не дозволяють окремому користувачу досягти бажаних результатів за прийнятний час.

Один з поширених методів захисту від злому за допомогою райдужних таблиць – використання необоротних хеш-функцій, які включають salt («сіль»). Існує безліч можливих схем змішування затравки і пароля. Наприклад, розглянемо таку функцію для створення хеш від пароля:

хеш = MD5(пароль + сіль)

Для такого відновлення пароля зловмиснику необхідні таблиці для всіх можливих значень солі. При використанні такої схеми, сіль повинна бути досить довгою (6-8 символів), інакше зловмисник може обчислити таблиці для кожного значення солі, випадкової і різною для кожного пароля. Таким чином, два однакових пароля будуть мати різні значення хешів, якщо тільки не буде використовуватися однакова сіль.

По суті, сіль збільшує довжину і, можливо, складність пароля. Якщо таблиця розрахована на деяку довжину або на деякий обмежений набір символів, то сіль може запобігти відновленню пароля. Наприклад, у старих Unix-паролях використовувалася сіль, розмір якої становив лише 12 біт. Для злому таких паролів зловмисникові потрібно було порахувати всього 4096 таблиць, які можна вільно зберігати на терабайтних жорстких дисках. Тому в сучасних програмах намагаються використовувати більш довгу сіль. Цей спосіб знижує ефективність застосування попередніх обчислень, тому що використання проміжних значень збільшує час, для обчислення одного пароля, і тим самим зменшує кількість обчислень, які зловмисник може провести у встановлені часові рамки.

Даний метод застосовується в наступних алгоритмах хешування: MD5, в якому використовується 1000 повторень, і bcrypt. Альтернативним варіантом є

використання посилення ключа (англ. key strengthening), який часто приймають за розтягнення ключа. Застосовуючи даний метод, ми збільшуємо розмір ключа за рахунок додавання випадкової солі, яка потім спокійно видаляється, на відміну від розтягування ключа, коли сіль зберігається і використовується в наступних ітераціях.

2.2. Індивідуальне завдання

Дослідити існуючі механізми зберігання пароля. Зробити порівняльну характеристику кожного механізму. Реалізувати механізм зберігання пароля та продемонструвати процес автентифікації. Довести, що даний метод оптимальний.

Контрольні запитання

1. Навіщо потрібне хешування паролів?
2. Наведіть приклади хеш-функцій.
3. Для чого призначений MD5?
4. Назвіть способи зберігання паролів.
5. Що таке лавинний ефект?
6. Які криптографічні бібліотеки підтримують MD5?
7. Що таке bcrypt?
8. Що таке SHA-1?
9. У чому відмінність SHA-1 від MD5?
10. Для чого використовується сіль?
11. Які є проблеми, пов’язані з сіллю?
12. Що таке райдужна таблиця?
13. У чому полягає головна ідея ланцюгів хешів?
14. У чому принцип роботи райдужних таблиць?
15. Опишіть принцип одного з методів захисту від злому за допомогою райдужних таблиць.

ЛАБОРАТОРНА РОБОТА 3

Time-Based one Time Password

Мета роботи: дослідити і реалізувати механізм генерації одноразових паролів TOTP.

3.1Теоретичні відомості

Одноразовий пароль (англ. one time password, OTP) – це пароль, який є дійсним тільки для одного сеансу автентифікації. Його дія також може бути обмежена певним проміжком часу. Перевага такого пароля порівняно зі статичним полягає в тому, що його неможливо використовувати повторно. Таким чином, зловмисник, що перехопив дані з успішної сесії автентифікації, не може використовувати скопійований пароль для отримання доступу до захищеної інформаційної системи. Використання одноразових паролів не захищає від атак, базованих на активному втручанні в канал зв'язку, що використовується для автентифікації (наприклад, від атак типу «людина посередині»).

Способи створення та розповсюдження OTP

Алгоритми створення OTP зазвичай використовують випадкові числа. Це необхідно, бо інакше було б легко передбачити подальші паролі на основі знання попередніх. Конкретні алгоритми OTP сильно розрізняються в деталях. Різні підходи до створення одноразових паролів перераховані нижче:

– Які використовують математичні алгоритми для створення нового пароля на основі попередніх (паролі фактично становлять ланцюжок, і повинні бути використані в певному порядку).

– Базовані на тимчасовій синхронізації між сервером і клієнтом, що забезпечує пароль (паролі дійсні протягом короткого періоду часу)

– Які використовують математичний алгоритм, де новий пароль базований на запиті (наприклад, випадкове число, яке обирає сервер, або частини вхідного повідомлення) та/або лічильнику.

Також існують різні способи повідомлення користувачеві наступного пароля.

– системи використовують спеціальні електронні токени, які користувач носить із собою й які створюють одноразові паролі і виводять потім їх на маленькому екрані;

– системи, що складаються з програм, які користувач запускає з мобільного телефону;

– системи генерують одноразові паролі на сервері і потім відправляють їх користувачеві, використовуючи сторонні канали, такі, як SMS – повідомлення;

– системи, у яких одноразові паролі надруковані на аркуші паперу або на скретч-картці, які користувачеві необхідно мати з собою.

Людина не в змозі запам'ятати одноразові паролі. Тому потрібні додаткові технології для їх коректної роботи.

Time-based One Time Password (TOTP) – це одноразовий код доступу, який зазвичай використовується для автентифікації користувачів. Користувачеві призначається генератор TOPT, що поставляється як апаратний брелок або програмний маркер. Генератор реалізує алгоритм, що обчислює одноразовий пароль, використовуючи секрет, який передається серверу автентифікації, і поточний час – звідси і назва OTP на основі часу. Пароль відображається користувачеві та діє протягом обмеженого періоду. Після закінчення терміну дії код доступу більше не діє. Користувач вводить дійсний пароль у форму для входу, як правило, разом із своїм ім’ям користувача та звичайним паролем.

Генератор/токени TOPT зазвичай використовуються як другий фактор автентифікації. Користувачам призначається апаратний маркер для генерації TOPT або програми, яка завантажується та прив'язується до мобільного пристрою або ПК.

TOPT був винайдений компанією RSA Security і продавався виключно під патентом до закінчення терміну дії патенту. Сьогодні рішення автентифікації TOPT були стандартизовані OATH та продаються багатьма постачальниками автентифікації.

Популярною альтернативою TOPT є OTP на основі подій, який також називають одноразовим паролем на основі HMAC/HOTP(HOTP)/HOTP він реалізує алгоритм, який обчислює одноразовий пароль, використовуючи секрет, спільний з сервером автентифікації та лічильником збільшується щоразу, коли створюється OTP (замість поточного часу в TOPT). TOPT і HOPT вважаються однаково безпечними, хоча деякі стверджують, що TOPT пропонує незначно кращу безпеку, оскільки паролі діють через певний проміжок часу, і зловмисник повинен використовувати викрадені паролі майже в режимі реального часу.

Історичні відомості

З 2004 року OATH (The Initiative for open authentication) працювала над проєктом одноразових паролів (OTP). Першим результатом був HOTP (the Hash-based Message Authentication Code (HMAC) OTP algorithm), опублікований у грудні 2005 року. Він був представлений як проект IETF (The Internet Engineering Task Force).

Подальша робота OATH йшла на поліпшення HOTP і в 2008 році був представлений TOTP. Цей алгоритм не використовує лічильник для синхронізації клієнта і сервера, а генерує пароль залежно від часу, який дійсний протягом деякого інтервалу. Алгоритм діє так: клієнт бере поточне значення таймера і секретний ключ, хеширує їх за допомогою якої-небудь хеш-функції і відправляє серверу, у свою чергу сервер проводить ті ж обчислення, після чого йому залишається тільки порівняти ці значення. Він може бути реалізований не тільки на хеш-функції SHA-1, на відміну від HOTP, тому хеш-функція також є вхідним параметром.

Пізніше, у вересні 2010 року був представлений новий алгоритм, що розширює TOTP ще більше, він отримав назву OATH Challenge-Response

Algorithms (OCRA). Головна відмінність від попередніх алгоритмів полягає в тому, що в перевірці достовірності бере участь також і сервер. Так що клієнт може бути впевнений у його достовірності.

Принцип роботи TOTP

По суті, TOTP є варіантом HOTP алгоритму, в якому як значення лічильника підставляється величина, що залежить від часу.

Основною відмінністю між двома алгоритмами є генерація пароля на основі мітки часу, яку використовують як параметра TOTP-алгоритм. При цьому використовується не точне значення часу, а поточний інтервал, межі якого були встановлені заздалегідь (наприклад, 30 с).

HOTP генерує ключ на основі розподіленого секрету і не залежного від часу лічильника. Модель цього алгоритму базована на події – наприклад, кожен раз, коли генерується черговий одноразовий пароль, лічильник буде збільшуватися. Отже, згенеровані згодом паролі повинні бути різними кожен раз.

Завдяки цьому основа для лічильника в HOTP алгоритмі, на відміну від інших алгоритмів, що використовують таймер, захищена від розсинхронізації передавальних пристроїв або занадто великої відстані між ними (такої відстані, коли відповідь від отримувача приходить пізніше, ніж закінчиться час валідності пароля). Це дозволяє HOTP-паролям залишатися дійсними протягом необмеженої кількості часу, в той час як TOTP-паролі перестануть бути дійсними через конкретний проміжок часу.

У підсумку, за умови використання тієї ж самої хеш-функції, як і в HOTP, дана відмінність у роботі алгоритму робить TOTP більш безпечним і кращим рішенням для одноразових паролів.

Розглянемо сам принцип роботи Time-based One Time Password

Позначимо:

– дискретне значення часу, що використовується як параметр. (вимірюється в одиницях, 4 байти);

– інтервал часу, протягом якого дійсний пароль (за замовчуванням 30 с.);

– початковий час, необхідний для синхронізації сторін. (за замовчуванням — час від початку UNIX ери);

– спільний секрет;

– поточний час.

Тоді

$$T = (CurrentTime - T_0)/X$$

$$HOTP(K, T) = Truncate(HMAC\text{-}SHA\text{-}1(K,T))$$

$$TOTP = HOTP(K, T)$$

де HMAC-SHA-1(K,T) – генерація 20-ти байт на основі таємного ключа і часу за допомогою хеш-функції *SHA-1*.

Truncate – функція вибору певним способом 4 байт:
позначимо *String* – результат *HMAC-SHA-1(K,T)*; *OffsetBits* – молодші 4 біта рядка *String*; *Offset = StringToNumber(OffsetBits)* і результатом *Truncate* буде рядок з чотирьох символів — *String[Offset]...String[Offset + 3]*

Також варто відзначити, що на відміну від *HOTP*, який базований тільки на *SHA-1*, TOTP може також використовувати *HMAC-SHA-256*, *HMAC-SHA-512* та інші *HMAC*-хеш-функції:

$$TOTP(K, T) = Truncate(HMAC\text{-}SHA\text{-}256(K,T))$$

$$TOTP(K, T) = Truncate(HMAC\text{-}SHA\text{-}512(K,T))$$

Реалізація в проєктах:

– компанія Google реалізувала версію TOTP в Google Authenticator;

– AWS також підтримує TOTP для входу AWS-консоль;

– Dropbox використала TOTP для доступу к серверу;

– LastPass підтримує TOTP використала Google Authenticator;

– бібліотека Liboath для створення як TOTP, так і HOTP паролів.

Розглядаючи надійність алгоритмів, можна сказати, що концепція одноразових паролів робить системи, що використовують ці алгоритми,

високонадійними. TOTP досить стійкий до криптографічних атак, проте ймовірності злому є, наприклад такий варіант атаки:

Оскільки пароль дійсний протягом деякого відрізка часу, то в теорії зловмисник може цим скористатися, "прослуховуючи" трафік клієнта і перехоплювати посланий логін і одноразовий пароль (або хеш від нього). Потім йому досить блокувати комп'ютер "жертви" і відправити автентифікаційні дані від власного імені. Якщо він встигне це зробити за інтервал часу, протягом якого дійсний пароль, то йому вдасться отримати доступ. Саме тому цей інтервал варто робити невеликим, проте і зовсім маленьким не варто, тому що у випадку невеликої розсинхронизації клієнт не зможе отримати доступ.

Також існує проблема, пов'язана із синхронізацією таймерів сервера і клієнта, оскільки існує ризик розсинхронізації інформації про час на сервері і в програмному та/або апаратному забезпеченні користувача. Оскільки TOTP використовує як параметра час, то при незбігу значень усі спроби користувача на автентифікацію закінчаться невдачею. У цьому випадку помилковий допуск чужого також буде неможливий. Варто зазначити, що ймовірність такої ситуації вкрай мала.

Google Authenticator

Google Authenticator – додаток для двоетапної автентифікації за допомогою Time-based One-time Password Algorithm (TOTP) і HMAC-based One-time Password Algorithm (HOTP) від Google.

Автентифікатор представляє 6-ти або 8-мизначний одноразовий цифровий пароль, який користувач повинен надати в додаток до імені користувача і пароля, щоб увійти в Google або інших сервісів. Автентифікатор також може генерувати коди для сторонніх додатків, такі як менеджери паролів або послуг хостингу файлів.

Як правило, користувачі повинні спочатку встановити програму на свій мобільний пристрій. Для того, щоб увійти на сайт або скористатися послугами

сервісу, потрібно ввести ім'я користувача та пароль, запустити додаток Authenticator і ввести в спеціальне поле згенерований одноразовий пароль.

Для цього, сайт надає загальний секретний ключ користувачеві, який повинен бути збережений у додаток Google Authenticator. Цей таємний ключ буде використовуватися для всіх майбутніх входів на сайт.

З двоетапною автентифікацією, просте знання логіна/пароля не є достатнім для злому облікового запису. Зловмисник також повинен знати секретний ключ або мати фізичний доступ до вашого пристрою з Google Authenticator. Альтернативним шляхом є MITM-атака: якщо комп'ютер заражений трояном, ім'я користувача, пароль і одноразовий код можуть бути перехоплені, щоб потім ініціювати свій власний сеанс входу на сайті або відслідковувати та змінювати інформацію між вами та сайтом.

Постачальник послуг генерує 80-бітний секретний ключ для кожного користувача. Це забезпечується як 16, 26, 32-значний код у кодуванні Base32 або за допомогою QR-коду. Клієнт створює HMAC-SHA1, використовуючи цей секретний ключ. Повідомлення HMAC може бути:

– числовим з 30-секундним періодом (TOTP);
– лічильником, який збільшується з кожним новим кодом (HOTP).

Потім частина HMAC витягується і перетворюється у 6-значний код.

Псевдокод для One Time Password OTP

```
function GoogleAuthenticatorCode(string secret)
    key := base32decode(secret)
    message := floor(current Unix time / 30)
    hash := HMAC-SHA1(key, message)
    offset := last nibble of hash
    truncatedHash := hash[offset..offset+3]  //4 bytes starting at the offset
    Set the first bit of truncatedHash to zero  //remove the most significant bit
    code := truncatedHash mod 1000000
    pad code with 0 until length of code is 6
```

return code

3.2. Індивідуальне завдання

Дослідити алгоритм Time-based One Time Password. Створити програму, що реалізує механізм генерації одноразових паролів TOTP. Для додаткових балів – організувати взаємодію з мобільним додатком Google Authenticator.

Контрольні запитання

1. Що таке одноразовий пароль?
2. Які існують підходи до створення одноразових паролів?
3. Які існують способи повідомлення користувачеві створеного пароля?
4. Для чого використовується алгоритм Time-based One Time Password?
5. Чим TOTP відрізняється від HOTP?
6. Який алгоритм за використання однакової хеш-функції є більш безпечним: TOTP чи HOTP?
7. Опишіть принцип роботи TOTP.
8. Які хеш-функції може використовувати TOTP? Які може використовувати HOTP?
9. Що виконує функція Truncate?
10. У яких проєктах реалізовано TOTP?
11. Які можливості злому TOTP та HOTP?
12. Які існують способи посилення захисту TOTP?
13. Що таке Google Authenticator?
14. Що потрібно для користування Google Authenticator?
15. Що потрібно для злому облікового запису з двоетапною автентифікацією?

ЛАБОРАТОРНА РОБОТА 4

Захист від зміни бінарного файлу

Мета роботи: навчитися підписувати виконувані файли.

4.1Теоретичні відомості

Цифровий підпис – це електронна зашифрована печатка, що засвідчує справжність цифрових даних, таких як повідомлення електронної пошти, макроси або електронні документи. Підпис підтверджує, що відомості надані, підписані їх творцем і не були змінені.

Цифровий підпис програми потрібен для того, щоб захистити програму за допомогою вказівки вашого авторства. Як тільки програма отримає спеціальний цифровий підпис, вона не може бути змінена третіми особами. Якщо людина спробує внести свої зміни в код програми, цифровий підпис тут же стане недійсним.

Додатки, що мають цифровий підпис, є верифікованими і не викликають підозр у користувачів. До них лояльно відносяться різні антивіруси і брандмауери. Такі програми дуже рідко потрапляють у карантин.

Основною проблемою сертифікації виконуваних файлів є той факт, що сам по собі підпис файлу, який свідчить про незмінність файлу, не гарантує його безпеки. Підпис показує лише, що файл ніким не був змінений після проходження процедури підписання. Авторитетність автора же центром сертифікації будь-яким чином не перевіряється. Більш того, приблизно 90 % потенційно небезпечних програм і 10 % шкідливих програм виявляються підписаними і успішно проходять перевірки сертифікатів.

Сертифікат підпису

Перше, що вам потрібно зробити, це отримати сертифікат і встановити його на свій комп'ютер.

Цифровий підпис підтверджує:

– Справжність. Цифровий підпис підтверджує особистість, підписала.

– Цілісність. Цифровий підпис підтверджує, що вміст документа не було змінено або підроблено після завірення.

– Невідворотність. Цифровий підпис підтверджує походження завіреного вмісту. Той, що підписав, не може заперечувати свій зв'язок з підписаним вмістом.

– Нотаріальне завірення. Підписи у файлах Microsoft Word, Microsoft Excel або Microsoft PowerPoint з відміткою захищеного сервера часу при певних обставинах рівносильні нотаріальному посвідченню.

Щоб підтвердити всі ці параметри, творець документа повинен засвідчити його вміст цифровим підписом, який задовольняє вказаним нижче вимогам.

– Цифровий підпис повинен бути дійсним.

– Сертифікат, пов'язаний з цифровим підписом, повинен бути чинним (Не простроченим).

– Фізична або юридична особа, що поставила цифровий підпис (видавець), має бути довіреною.

Сертифікат можна або купити у Центрі сертифікації, або створити його за допомогою makecert. Розглянемо обидва варіанти.

Центр сертифікації (Certificate Authority). Центр сертифікації схожий на нотаріальну контору. Він випускає цифрові сертифікати, підтверджує їх достовірність за допомогою підписів, а також відстежує сертифікати, які минули або були відкликані.

Щоб створити цифровий підпис, потрібен сертифікат підпису, що засвідчує особу. Разом з макросом або документом, завіреним підписом, також відправляється сертифікат і відкритий ключ. Сертифікати випускаються центром сертифікації і, аналогічно посвідченню водія, можуть бути відкликані. Як правило, сертифікат дійсний протягом року, після закінчення якого підписує

повинен продовжити його або отримати новий сертифікат для посвідчення своєї особистості.

Переваги та недоліки покупки сертифікату у центрі сертифікації:

+ Використовуючи сертифікат, виданий центром сертифікації, можна бути впевненим, що система Windows не буде попереджувати кінцевого користувача про програму від "unknown publisher" на будь-якому комп'ютері, що використовує сертифікат від CA.

– Сертифікат від Certificate Authority не є безкоштовним.

Створення сертифіката за допомогою Makecert

MakeCert – інструмент для створення сертифікатів X.509, які призначені виключно для тестування розробки. Цей інструмент створює пару ключів (відкритий і закритий) для цифрового підпису та поміщає її в файл сертифіката. MakeCert входить до складу пакета Windows SDK.

MakeCert – це консольний застосунок. Для роботи з ним необхідно запустити командний рядок (від імені Адміністратора) і ввести команду запуску програми з зазначеними параметрами: *makecert [options] outputCertificateFile*, де *outputCertificateFile* – ім'я файлу з розширенням .cer, в який буде записаний тестовий сертифікат X.509, *options* – параметри створення сертифіката.

Створення самопідписаного сертифіката:

Виконуємо в командному рядку:

makecert -n "CN = TempCert" -r -sv TempCert.pvk TempCert.cer

Створення самопідписаного сертифіката

-n (subjectName) – задає ім'я суб'єкта. Згідно з правилами, до імені суб'єкта додається префікс "CN =" для "Common Name";

-r – вказує, що сертифікат самопідписаний;

-sv (privateKeyFile) – вказує файл, який містить контейнер закритого ключа. Тобто закритий ключ буде зберігатися не в сертифікаті, а у файлі.

Самопідписаний сертифікат – це сертифікат, підписаний застосунком, який створив його.

Щоб створити тимчасовий сертифікат, необхідно виконати дві дії. Перше – створити самопідписаний сертифікат, який буде використовуватися як кореневий сертифікат для тимчасового сертифіката. Друге – створити сам тимчасовий сертифікат, підписаний кореневим сертифікатом.

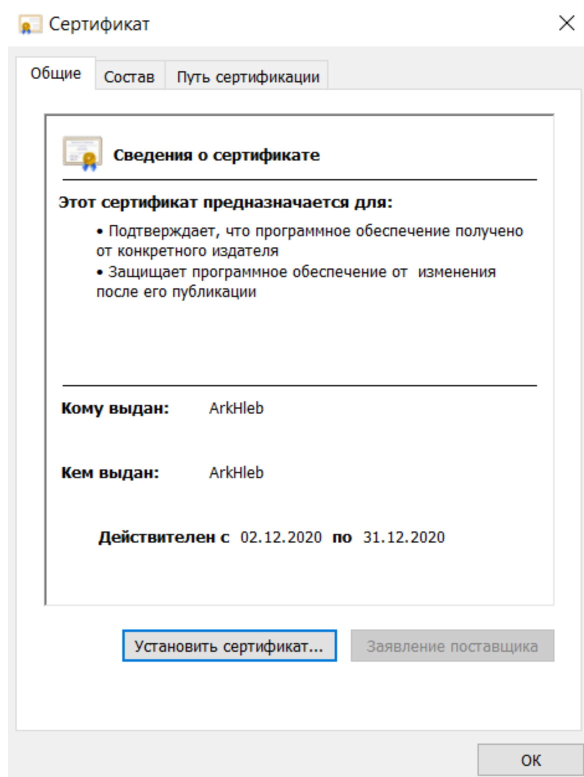


Рисунок 4.1 – Сертифікат, створений за допомогою Makecert

Далі потрібно зробити створений сертифікат довіреним, додавши його до відповідного списку сертифікатів на своєму ПК (рис. 4.1).

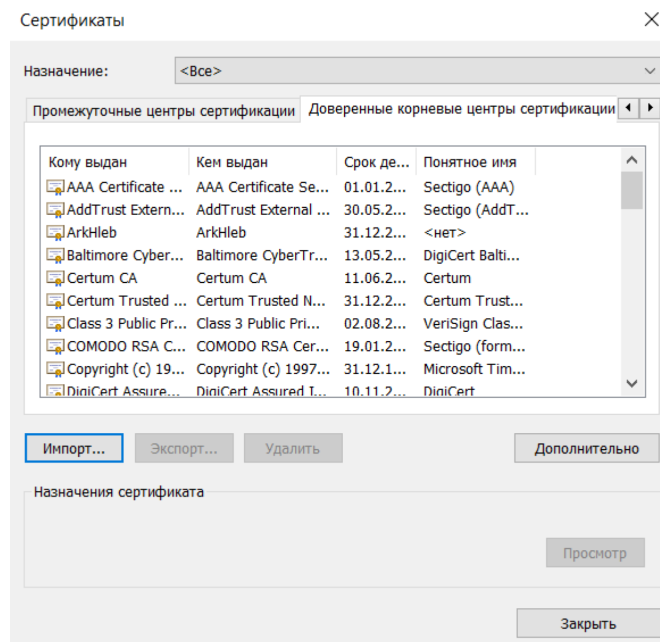


Рисунок 4.2 – Список довірених сертифікатів

Переваги та недоліки створення сертифіката за допомогою Makecert:

+ Створення сертифіката за допомогою Makecert не викликає особливих проблем і крім того, можна поділитися сертифікатом з кінцевими користувачами.

– Кінцеві користувачі повинні будуть вручну встановити сертифікат на своїх комп'ютерах і залежно від ваших клієнтів це може бути не варіант.

– Сертифікати, створені за допомогою makecert, зазвичай використовуються для розробки і тестування, а не для виробництва.

Підписування бінарних файлів

Існують різні утиліти для підпису програмного забезпечення. Розглянемо один з таких інструментів, а саме: «signtool.exe» від Microsoft.

SignTool – це засіб командного рядка, що використовується для цифрового підпису пакета додатка або пакета застосунків за допомогою сертифіката. Сертифікат може бути створений користувачем (для тестування) або виданий компанією (для поширення). Підписування застосунку дає користувачеві засіб перевірки відсутності змін в даних застосунка після його підпису, при цьому також підтверджується справжність користувача або компанії, яка підписала

застосунок. SignTool також може підписати зашифровані і незашифровані пакети застосунку і пакети застосунків.

Синтаксис

signtool [command] [options] [file_name | ...]

Параметри

command – одна з чотирьох команд (catdb, sign, Timestamp або Verify), яка визначає операцію, яку слід виконати над файлом. Опис кожної команди дивись у наступній таблиці.

options – опція, яка змінює команду. На додаток до глобальних параметрів /q та /v, кожна команда підтримує унікальний набір параметрів.

file_name – шлях до файлу для підписання.

Наступні команди підтримуються програмою SignTool. Кожна команда використовується з різними наборами опцій.

Для підпису за допомогою SignTool.exe необхідно мати:

– застосунок;
– дійсний сертифікат підпису;
– SignTool.exe.

SignTool може використовуватися для підписування файлів, перевірки підписів і міток часу, видалення підписів та іншого. Оскільки нас цікавить підписування застосунку, ми розглянемо команду sign.

Наступна команда підписує файл за допомогою сертифіката, що зберігається у захищеному паролем файлі PFX:

SignTool sign /f MyCert.pfx /p MyPassword MyControl.exe

SignTool повертає текст командного рядка, який вказує результат операції підписання. Крім того, SignTool повертає вихідний код нуля для успішного

виконання, одного для невдалого виконання та двох для виконання, яке завершено попередженнями.

Після підписування файлу за допомогою SignTool (рис. 4.3) можна перевірити справжність сертифіката, яким підписано файл, дату підпису та ім’я того, хто підписав, також можна дізнатися алгоритм шифрування, серійний номер та інші параметри.

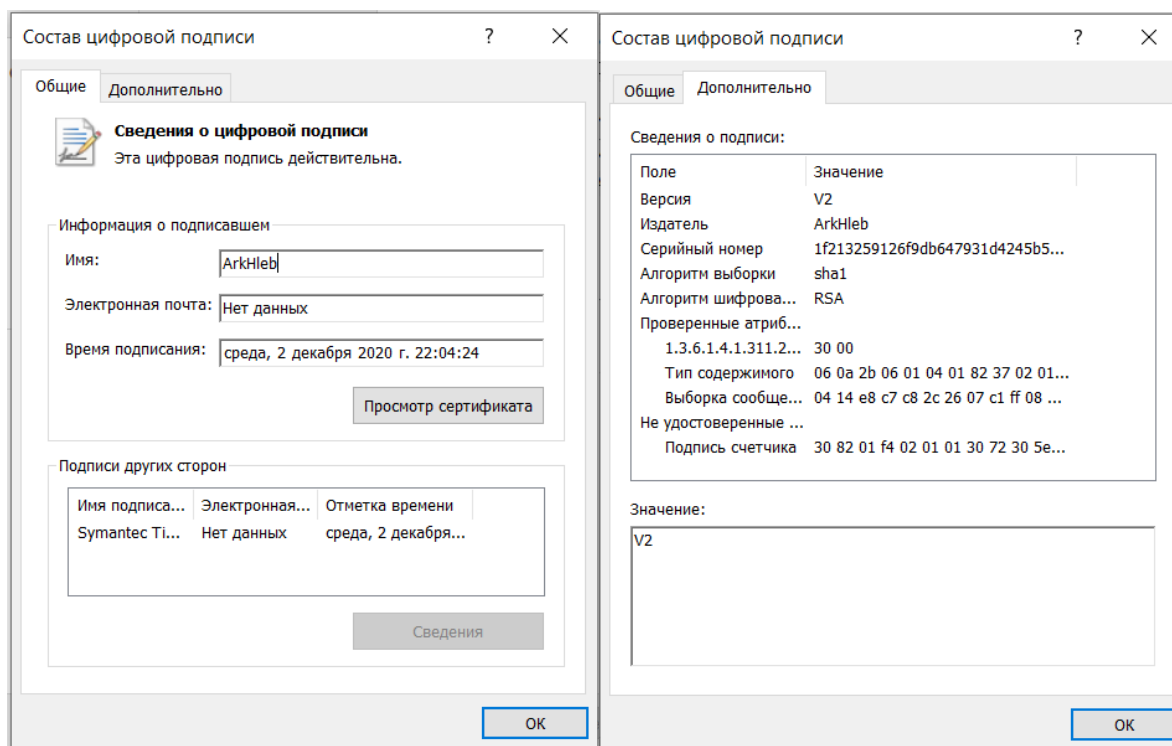


Рисунок 4.3 – Інформація про цифровий підпис

4.2. Індивідуальне завдання

1. Створити сертифікат.
2. Проінсталювати його в систему, щоб він був довіреним.
3. Використовуючи проєкт будь-якої попередньої роботи, виконати підпис виконуваного файлу за допомогою утиліти SignTool (або JarSigner) (інші варіанти повинні бути оговорені з викладачем).
4. Виконати верифікацію підпису (бажано на рівні самого коду при завантаженні застосунку):

4.1Чи є підписаний сертифікат валідним.

4.2Чи не було (бінарної) зміни файлу та чи його код цілісний.

Контрольні запитання

1. Що таке цифровий підпис?
2. Навіщо потрібен цифровий підпис?
3. У чому полягає основна проблема сертифікації?
4. Що підтверджує цифровий підпис?
5. Які існують вимоги до цифрового підпису?
6. Назвіть переваги та недоліки сертифікатів, що були видані центром сертифікації.
7. Назвіть переваги та недоліки самопідписаних сертифікатів.
8. Що таке MakeCert?
9. Що потрібно для того, щоб створити сертифікат за допомогою MakeCert?
10. Як зробити створений сертифікат довіреним?
11. Що таке SignTool?
12. За допомогою якої команди виконується підпис файлу у SignTool?
13. Що потрібно мати для підписування файлу за допомогою SignTool?
14. Які існують альтернативи SignTool?
15. Як можна виконати верифікацію підпису?

ЛАБОРАТОРНА РОБОТА 5

Геш-дерева

Мета роботи: ознайомитись з технологією Merkle Tree.

5.1.Теоретичні відомості

Дерево Меркла (хеш-дерево, Merkle Tree) представляє собою особливу структуру даних, яка містить підсумкову інформацію про деяким більший обсяг даних. Використовується для перевірки цілісності даних.

Дерева Меркла є корисною складовою багатьох технологій (здебільшого тих, що розподіляються в архітектурі), і заслуга в розробленні такої чудової концепції належить Ральфу Меркелю. У 1979 році Ральф Меркл запатентував концепцію хеш-дерев, а в 1987 році опублікував статтю "Цифровий підпис, базовану на звичайній функції шифрування", яка використовувала цю концепцію. Хоча термін його дії патенту закінчився у 2002 році, ця структура даних стала настільки корисною частиною різних технологій, що згодом її стали називати деревами меркла.

Інтернет розвивався майже два десятиліття, і ділитися вашим вмістом в Інтернеті було не так просто, як зараз. З часом було створено багато різних служб, і тепер люди можуть легко ділитися своїм вмістом в Інтернеті. Будь то платформи для обміну відео, такі як YouTube, платформи для обміну думками, такі як Twitter, або навіть для створення власних сайтів, такі як Wordpress.

Ці послуги покращили доступність Інтернету, але є також певні занепокоєння щодо такого роду структури Інтернету, які почали виникати. Деякі з них:

– Концентрація даних: Дані, які ми генеруємо, або інформація, яка нам потрібна для доступу в Інтернеті, постійно накопичуються в центрах обробки даних кількох гігантських компаній (які надавали всі ці послуги хостингу

контенту взамін), і зненацька ми зрозуміли, що мати контроль над нашими даними так само важливо, як і надавати послуги, які пропонують ці компанії.

– Адресація на основі хоста: У перші дні Інтернету, коли такі послуги, як Facebook, Blogspot, Pinterest тощо, були відсутні, єдиним способом поділитися тим, що ви хочете, було закрутити сервер, отримати домен та розмістити вміст . Потім ми можемо поділитися своїм вмістом, поділившись адресою.

Зараз існують соціальні медіа-платформи, де люди діляться своєю інформацією, і адреси тепер є чимось на зразок your_site.com/your_content та your_site.com/someone_elses_content, що призводить до небажаного контролю, концентрації і навіть дублювання вмісту. Отже, необхідність кращого способу звернення є очевидною, що може відокремити хост від самого вмісту, тобто нам потрібна адресація на основі вмісту.

Неможливо скористатися перевагами мережі від близькості: Веб-сервери розміщеного веб-сайту можуть обслуговуватися з далекої частини земної кулі. Крім того, можна одночасно завантажувати один вміст лише з одного сервера. Набагато ефективнішим способом отримання вмісту може бути отримання декількох його фрагментів із кількох сусідніх комп'ютерів у мережі, які вже мають частини вмісту, який ми шукаємо.

Потрібен був механізм, щоб перевірити, що учасники мережі не мають:

– випадково пошкоджених даних;

– навмисно фальсифікованих даних.

Одним із способів вирішення цієї проблеми можуть стати криптографічні хеш-функції.

Цей підхід спрацював би, але він не ефективний через такі причини:

– Перевірка лише після завершення: пір повинен чекати, поки надійдуть цілі дані (від кількох пірів), перш ніж він зможе перевірити їх справжність. Якщо тисячі пірів розкидані по всьому світу (як у торрент-мережі), частина фрагментів прибуде раніше, а частина – ні. Отже, очікування отримання всіх даних перед тим, як перевірити, чи правильні вони, насамперед, не є ефективним підходом.

– Неможливо з’ясувати винуватця: якщо перевірка не вдається (хеш, наданий надійним сервером, не збігається з хешем файлу, завантаженого рівноправним користувачем), неможливо з’ясувати, які піри відповідали за надсилання неправильного фрагменту.

– Занадто багато накладних витрат на синхронізацію: Скажімо, ми хочемо оновити вміст нашого існуючого файлу. Навіть якщо в якомусь фрагменті є одна зміна символів, весь файл потрібно знову хешувати, а генерований хеш потрібно повідомити довіреному серверу.

– Забагато "довіри" на довіреному сервері: Якщо сервер був скомпрометований, неможливо дізнатись, чи проблема з пірами, чи з надійним сервером, чи у всіх.

Звичайно, є альтернативний варіант. Існує можливість замість того, щоб просто зберігати хеш повного файлу, підтримувати хеші окремих фрагментів на "надійному" сервері. Таким чином, можна перевірити фрагменти, збираючи їх у ненадійних пірів.

Це вирішить проблему перевірки лише після завершення та неможливості з’ясувати винуватого, оскільки буде можливість перевірити фрагменти окремо, як тільки вони будуть завантажуватися від пірів. Це також може дещо вирішити проблему з занадто великою кількістю накладних витрат на синхронізацію, оскільки нещодавно оновлені фрагменти можна хешувати окремо, і цю інформацію можна передавати надійному серверу, але:

– довірений сервер повинен зберігати хеші, що відповідають всім фрагментам, які відповідають всім файлам, що означає більшу кількість вимог до зберігання, ніж у попередньому випадку;

– імовірність хеш-зіткнення дещо зростає;

– щоб перевірити кожен завантажений фрагмент, потрібно покластися на надійний сервер. Все ще нічого не можна зробити, якщо надійний сервер буде скомпрометований.

Розглянемо хеш-ланцюжок як можливе рішення (рис. 5.1).

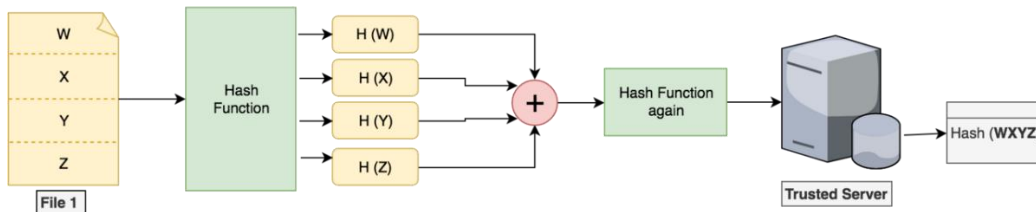


Рисунок 5.1 – хеш-ланцюжок

Якщо об'єднати всі окремі хеші та знову їх хешувати, щоб отримати кореневий хеш, то потрібно зберегти цей кореневий хеш у надійному місці. Можна використовувати цей кореневий хеш для адресації та перевірки на основі вмісту, отже:

– Тільки на початку, пір може збирати хеші всіх фрагментів від інших пірів у мережі (які містять фрагменти необхідного файлу).
– Після завантаження будь-якого фрагмента, пір може обчислити свій хеш, а потім обчислити кореневий хеш і перевірити, чи існує він на надійному сервері.
– Не потрібно турбуватися про те, щоб заважати довіреному серверу, і можна перевірити майбутні фрагменти локально, що зменшить надмірну залежність довіреного сервера.

Все ще нічого не можна зробити, якщо сервер скомпрометований і дає помилкове підтвердження існування хешу.

Зберігання додаткової інформації на надійному сервері

Що робити, якщо як окремі хеш-фрагменти, так і кореневий хеш зберігаються на сервері? За рахунок більшої кількості місця на довіреному сервері, нарешті, вирішена проблема перевірки надійності сервера, за допомогою виконання процесу для перевірки:

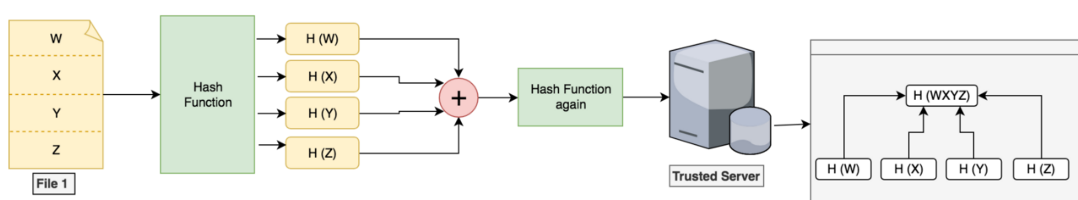


Рисунок 5.2 – хеш-ланцюжок зберігаэться на сервері

Замість того, щоб надсилати підтвердження існування, сервер надсилає весь доказ рівноправній особі, що вимагає підтвердження. Під цілим доказом маються на увазі всі хеші побратимів та кореневого фрагмента.

Оскільки сервер зараз надсилає доказ, який може перевірити клієнт, клієнт тепер обчислює кореневий хеш шляхом об'єднання:

- геш поточного фрагмента, який він / вона хоче перевірити;
- геші всіх колег, отриманих від надійного сервера.

Потім, нарешті, обчислюється кореневий хеш і перевіряється з кореневим хешем, який використовується для пошуку файлу в мережі.

Merkle Tree

Що робити, якщо ми зберігаємо дерево (рис. 5.3) на довіреному сервері?

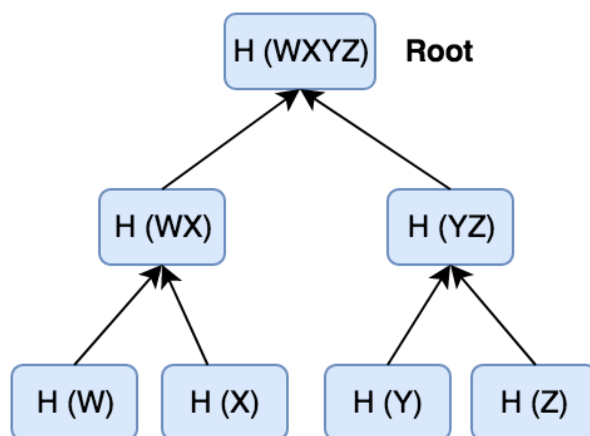


Рисунок 5.3-дерево довіреного серверу

Листя дерева відповідають хешам фрагментів даних файлу, а батьки цього листя є хешами конкатенації.

Саме такі структури називаються деревами Меркла (також хеш-деревами).

Кореневий геш використовується для адресації на основі вмісту. Деревоподібна організація merkle дозволяє виконувати наступні завдання дуже ефективно з точки зору зберігання та обчислень:

1. Перевірка даних

Ось як відбувається перевірка даних у Merkle tree:

– завантажується частина даних із ненадійної мережі;

– виконується запит до сервера надати підтвердження того, що цей фрагмент знаходиться у дереві;

– сервер повертає відповідні хеші;

– за допомогою цієї інформації, обчислюється кореневий хеш і порівнюється з кореневим хешем, за допомогою якого було отримано доступ до файлу.

Наприклад, якщо пір хоче перевірити, чи фрагмент "Y" існує у файлі та не макетований, сервер повертає інформацію H (Z) та H (WX), яка також має назву аудиторський слід.

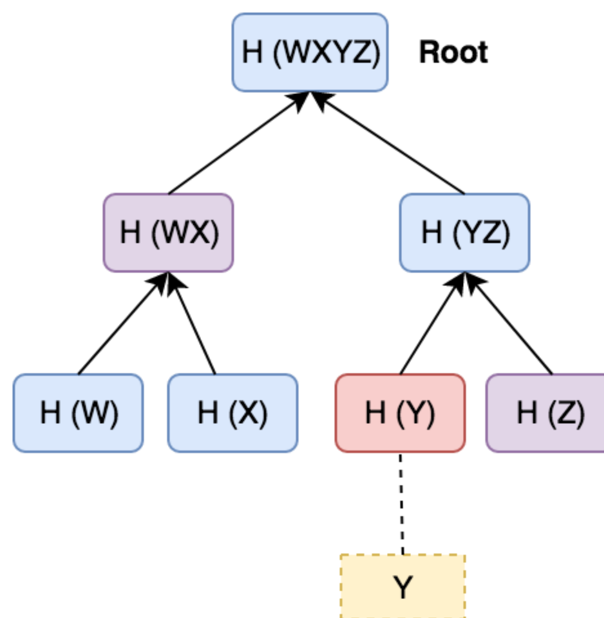


Рисунок 5.4 – Дерево даних

Потім можна обчислити:

H(YZ) від H(Y), який було обчислено, та H(Z), який було надано надійним сервером.

H(WXYZ) від H(YZ) , який було обчислено, та H (WX), який було надано надійним сервером.

Нарешті, можна порівняти H(WXYZ), обчислений з вихідним кореневим хешем, який використовується для пошуку файлу в ненадійній мережі. Якщо хеші

збігаються, доказ підтверджує, що фрагмент існує в дереві і не підроблений / пошкоджений. Якщо перевірка не вдасться, можна заблокувати цей аналог і попросити той самий фрагмент у іншого піра, який має необхідний файл. Цей процес також відомий як аудиторське підтвердження.

Для перевірки даних від надійного сервера було потрібно дуже мало інформації. Якщо кількість фрагментів даних подвоїться, додатковою інформацією, необхідною для перевірки, буде лише ще два хеші, а для перевірки на стороні клієнта потрібні ще два хеш-обчислення.

Оскільки розмір пакета для перевірки даних невеликий, це допомагає економити пропускну здатність.

Це було значним покращенням порівняно з попереднім підходом геш-ланцюжка, коли довіреному серверу довелося надсилати всі геші пірів-пірів, а рівноправний код використовував їх об'єднання для обчислення кореневого гешу.

2. Перевірка узгодженості

Перевірка узгодженості бажана в системах, що ведуть незмінний журнал даних. Застосовується для перевірки того, що весь журнал є незамінтованим, що означає перевірку того, що нова версія у будь-який часовий проміжок включає всі дані попередньої версії та в тому ж порядку.

3. Синхронізація даних

Merkle tree можна використовувати для синхронізації даних між кількома вузлами в розподіленій системі. Для Merkle tree не потрібно порівнювати всі дані, щоб зрозуміти, що змінилося – достатньо просто зробити хеш-порівняння дерев. Після того, як буде з’ясовано, які листки були змінені, відповідний фрагмент даних може бути надісланий по мережі та синхронізований по всіх вузлах.

Використання Merkle tree

Merkle tree можуть бути невід’ємною частиною систем, які вимагають:

– перевірки даних;

– перевірки узгодженості;

– синхронізації даних.

Прикладом таких систем можуть бути:

Криптовалюти

Багато криптовалют (включаючи біткойни) зберігають дані транзакцій у структурі дерева merkle. Дерева Меркле допомагають перевірити узгодженість, тобто переконатися, що нова версія книги включає всі транзакції з попередньої версії в тому ж порядку.

Системи контролю версій

Системи контролю версій, такі як Git та Mercurial, використовують спеціалізовані дерева Merkle для управління версіями файлів і навіть каталогів. Однією з переваг використання дерев Merkle в системах контролю версій є те, що ми можемо просто порівнювати хеші файлів і каталогів між двома комітами, щоб знати, чи були вони змінені чи ні, що досить швидко.

Органи сертифікації

Органи сертифікації використовують дерево Merkle у своїх механізмах для ведення журналів прозорості сертифікатів, які можна перевірити. Журнал продовжує зростати, оскільки до нього додаються нові сертифікати, а прозорість перевіряється за допомогою механізму перевірки журналу.

Системи баз даних

Системи розподілених баз даних No-SQL, такі як Apache Cassandra та Amazon DynamoDB, використовують дерева міркувань для виявлення невідповідностей між репліками даних. Цей процес відновлення даних шляхом порівняння всіх реплік та оновлення кожної з них до останньої версії також називається відновленням проти ентропії. Процес також описаний у документації Apache Cassandra.

5.2. Індивідуальне завдання

Створити екосистему (та продемонструвати її роботу), що складається з наступних компонентів:

1. Сервер. Має інформацію про файли мережі у такому вигляді:
   - merkle root
   - перелік клієнтів, що мають хоча б частину контенту файлу та перелік блоків, які вони мають.
2. Клієнт, що має хоча б частину контенту файлу. Дії, що можна проводити над клієнтом:
   - запитати частину файла. На вхід подається merkle root цього файлу, індекс блоку. На вихід дається контент файлу або помилка, якщо такого блоку немає;
   - виконати верифікацію блока. На вхід подається merkle root, block hash. На вихід подається обрізане під-дерево (гілку до запитаного хеша) стосовно алгоритму. Якщо block hash відсутній або невалідний – повертається стосовна помилка.
3. Клієнт, що завантажує файл. Послідовність дій, що виконуються:
   - отримує будь-яким чином merkle root бажаного для завантажування файлу;
   - виконує завантаження блоку:
     - питає у сервера перелік клієнтів, що має певну частину цього файлу.
     - завантажує блок;
   - виконує верифікацію блоку:
     - виконує хешування блоку;
     - обирає будь-який інший сервер, що має цей блок, та запитує його частину merkle дерева. Якщо іншого сервера немає, питаємо у того, з якого завантажували;
     - самостійно проводить верифікацію гілки дерева;

- якщо усе добре – зберігає блок, оновлює внутрішню базу даних, посилає запит на сервер для додання запису, що даний клієнт має блок з таким індексом для певного файлу (merkle root).

Обмеження:

– розмір блоку файлу фіксований протягом усього життєвого циклу додатка, наприклад, 20Кб;

– кожен клієнт має у себе базу даних MongoDB, в якій є перелік розташування файлів та поточний хості на merkle дерево його контенту. При завантаженні кожного блоку дерево поповнюється автоматично.

Контрольні запитання

1. Що таке Merkle Tree?

2. Чому Merkle Tree має таку назву?

3. Які проблеми виникли разом із розвитком Інтернету?

4. Чому криптографічні хеш-функції не є оптимальним способом перевірки цілісності даних в Інтернеті?

5. Чому очікування отримання всіх даних перед тим, як перевірити, чи правильні вони, не є ефективним підходом?

6. Які проблеми вирішить зберігання хешів фрагментів інформації на сервері?

7. Які є недоліки такого методу?

8. Назвіть переваги та недоліки використання хеш-ланцюгів.

9. Як відбувається перевірка даних у Merkle Tree?

10. Що таке аудиторське підтвердження?

11. Чим використання Merkle Tree краще хеш-ланцюгів?

12. Навіщо проводити перевірку узгодженості?

13. Що потрібно для синхронізації даних за використання Merkle Tree?

14. Наведіть приклади систем, у яких можуть бути використані Merkle Tree?

15. У чому переваги використання Merkle Tree для систем контролю версій?

ЛАБОРАТОРНА РОБОТА 6

Алгоритм шифрування *RSA*

Мета роботи: дослідити і реалізувати механізм асиметричного алгоритму шифрування RSA на основі згенерованих ключів.

6.1. Теоретичні відомості

Типи шифрування

Існує два основних типи шифрування – симетричне і асиметричне.

Симетричне шифрування використовує один криптографічний ключ для шифрування і дешифрування даних. Найбільш видатною особливістю симетричного шифрування є простота процесу, тому що використовується один ключ як для шифрування, так і для дешифрування. Там, де необхідно зашифрувати великий об'єм даних, симетричне шифрування виявляється відмінним варіантом.

В основному, симетричні алгоритми шифрування вимагають менше обчислень, ніж асиметричні. На практиці, це означає, що якісні асиметричні алгоритми в сотні або в тисячі разів повільніші за якісні симетричні алгоритми. Недоліком симетричних алгоритмів є необхідність мати секретний ключ з обох боків передачі інформації. Оскільки ключі є предметом можливого перехоплення, їх необхідно часто змінювати та передавати по безпечних каналах передачі інформації під час розповсюдження. Існують сотні алгоритмів симетричного типу. Найбільш поширені з них – AES, RC4, DES, 3DES, RC5, RC6 і т. д.

Переваги:

- швидкість (за даними Applied Cryptography – на 3 порядки вище);
- простота реалізації (за рахунок більш простих операцій);
- необхідна менша довжина ключа для порівнювальної стійкості;
- вивченість (за рахунок більшого віку).

Недоліки:

– Складність управління ключами у великій мережі. Це означає квадратичне зростання числа пар ключів, які треба генерувати, передавати, зберігати і знищувати в мережі. Для мережі в 10 абонентів потрібно 45 ключів, для 100 вже 4950 і т. д.

– Складність обміну ключами. Для застосування необхідно вирішити проблему надійної передачі ключів кожному абоненту, тому що потрібен секретний канал для передачі кожного ключа обом сторонам.

Асиметричне шифрування, на відміну від симетричного, включає в себе кілька ключів для шифрування і дешифрування даних, які математично пов'язані один з одним. Один з цих ключів відомий як «відкритий ключ», а інший – як «закритий ключ». Асиметричний метод шифрування також відомий як «криптографія з відкритим ключем».

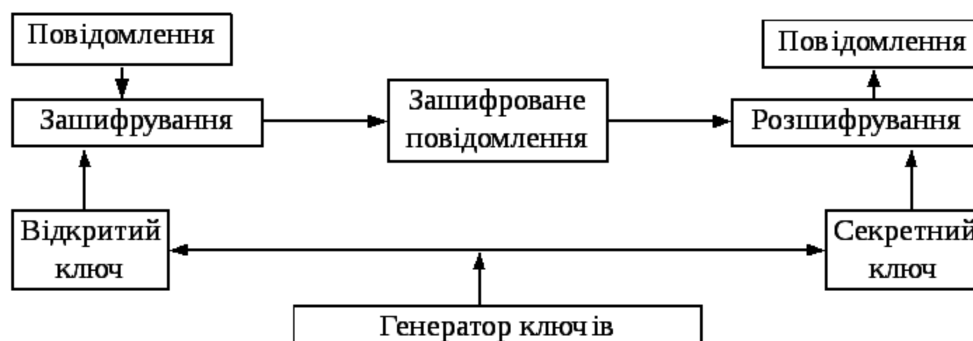


Рисунок 6.1 – Схематичне зображення асиметричного шифрування

Перша (і найбільш очевидна) перевага цього типу шифрування – безпека, яку він забезпечує. У цьому методі відкритий ключ – який є загальнодоступним – використовується для шифрування даних, в той час як розшифрування даних виконується з використанням закритого ключа, який необхідно надійно зберігати. Це гарантує, що дані залишаються захищеними від атак «людини посередині» (MiTM). Для веб-серверів і серверів електронної пошти, які постійно

підключаються до сотень тисяч клієнтів, потрібно управляти тільки одним ключем і захищати його. Інший ключовий момент полягає в тому, що криптографія з відкритим ключем дозволяє створювати зашифроване з'єднання без необхідності зустрічатися в автономному режимі, щоб спочатку обмінятися ключами.

Друга важлива особливість, яку пропонує асиметричне шифрування, – це аутентифікація. Дані, зашифровані за допомогою відкритого ключа, можуть бути розшифровані тільки за допомогою закритого ключа, пов'язаного з ним. Отже, він гарантує, що дані бачить і дешифрує тільки той об'єкт, який повинен їх отримати. Простіше кажучи, це підтверджує, що ви розмовляєте або обмінюєтеся інформацією з реальною людиною або організацією.

Найвідомішими асиметричними криптографічними системами є системи RSA (Rivest, Shamir, Adleman), Діффі-Хелмана, Эль-Гамаля і криптосистема на основі еліптичних кривих.

Переваги асиметричних шифрів перед симетричними:

– не потрібно попередньо передавати особистий ключ по надійному каналу;

– тільки одній стороні відомий ключ дешифрування, який потрібно тримати в секреті (у симетричній криптографії такий ключ відомий обом сторонам і повинен триматися в секреті обома);

– у великих мережах число ключів в асиметричній криптосистемі значно менше, ніж в симетричній.

Недоліки алгоритму несиметричного шифрування в порівнянні з симетричним:

– в алгоритм складніше внести зміни;

– довші ключі;

– шифрування-розшифрування з використанням пари ключів проходить на два-три порядки повільніше, ніж шифрування-розшифрування того ж тексту симетричним алгоритмом;

– потрібні істотно більші обчислювальні ресурси, тому на практиці асиметричні криптосистеми використовуються в поєднанні з іншими алгоритмами:

- для ЕЦП повідомлення попередньо піддається хешуванню, а за допомогою асиметричного ключа підписується лише відносно невеликий результат хеш-функції;
- для шифрування вони використовуються в формі гібридних криптосистем, де великі обсяги даних шифруються симетричним шифром на сеансовому ключі, а за допомогою асиметричного шифру передається тільки сам сеансовий ключ.

Алгоритм шифрування RSA

RSA – криптографічний асиметричний алгоритм з відкритим ключем. Безпека алгоритму RSA побудована на принципі складності факторизації цілих чисел. Алгоритм використовує два ключі – відкритий (*public*) і секретний (*private*), разом відкритий і відповідний йому секретний ключі утворюють пари ключів (*keypair*). Відкритий ключ не потрібно зберігати в таємниці, він використовується для шифрування даних. Якщо повідомлення було зашифровано відкритим ключем, то розшифрувати його можна тільки відповідним секретним ключем.

Алгоритм RSA складається з 4 етапів: генерації ключів, шифрування, розшифрування та розповсюдження ключів.

Генерація ключів

Для того, щоб згенерувати пари ключів, виконуються такі дії:

1. Вибираються два великі прості числа p і q приблизно 512 біт завдовжки кожне.

2. Обчислюється їх добуток $n = pq$;

3. Обчислюється функція Ейлера $j(n) = (p-1)(q-1)$;

4. Вибирається ціле число e таке, що $1 < e < j(n)$ та e взаємно просте з $\varphi(n)$;

5. За допомогою розширеного алгоритму Евкліда знаходиться число d таке, що $edº1\ mod(j(n))$.

Число n називається модулем, а числа e, d – відкритою й секретною експонентами (англ. *encryption* and *decryption exponents*), відповідно. Пари чисел (n, e) є відкритою частиною ключа, а (n, d) – секретною. Числа p і q після генерації пари ключів можуть бути знищені, але в жодному разі не повинні бути розкриті.

Шифрування

Припустимо, що Боб хотів би відправити повідомлення M Алісі. Спочатку він перетворює M в ціле число m так, щоб $0 \leq m < n$ за допомогою узгодженого оборотного протоколу, відомого як схеми доповнення. Потім він обчислює зашифрований текст c, використовуючи відкритий ключ Аліси e, за допомогою рівняння $c = m^e\ mod(n)$.

Це може бути зроблено досить швидко, навіть для 500-бітних чисел, з використанням модульного зведення в ступінь. Потім Боб передає c Алісі.

Розшифрування

Для розшифрування повідомлення Боба m Алісі потрібно обчислити таку рівність:

$$m = c^d \bmod n$$

Неважко переконатися, що при розшифруванні відновиться вихідне повідомлення:

$$c^dº(m^e)^dºm^{ed}(\bmod n).$$

З умови

$$edº1\ (\bmod j(n))$$

випливає, що

$$ed = kj(n) + 1 \text{ для деякого цілого } k,$$

$$\text{отже} m^{ed}ºm^{kj(n)+1}\ (\bmod n)$$

Згідно з теоремою Ейлера:

$$m^{j(n)}º1\ (\bmod n),$$

тому

$$m^{kj(n)+1} \equiv m \pmod{n}$$

$$c^{d} \equiv m \pmod{n}$$

Розповсюдження ключів

Для того, щоб Боб міг відправити свої секретні повідомлення, Аліса передає свій відкритий ключ (n, e) Бобу через надійний, але не обов'язково секретний маршрут. Секретний ключ d ніколи не розповсюджується.

Наведений вище варіант шифрування називається RSA з підручника (англ. textbook RSA) і є цілком уразливим. В жодному разі його не можна використовувати в криптосистемах.

Таблиця 6.1 – Приклад виконання алгоритму

Етап	Опис операції	Результат операції
Генерація ключів	Обрати два простих різних числа	$p = 3557$, $q = 2579$
	Обчислити добуток	$n = p * q = 3557 * 2579 = 9173503$
	Обчислити функцію Ейлера	$\varphi(n) = (p-1)(q-1) = 9167368$
	Обрати відкриту експоненту	$e = 3$
	Обчислити секретну експоненту	$d = e^{-1}(\text{mod } \varphi(n))$ $d = 6111579$
	Опублікувати відкритий ключ	$\{e, n\} = \{3, 9173503\}$
	Зберегти секретний ключ	$\{d, n\} = \{611579, 9173503\}$
Шифрування	Обрати текст для шифрування	$m = 111111$
	Обчислити шифротекст	$c = E(m) = m^{e} \text{ mod } n$

		$= 111111^3 \bmod 9173503$ $= 4051753$
Розшифрування	Обчислити вихідне повідомлення	$m = D(c) = c^d \bmod n$ $= 4051753^{6111579} \bmod 9173503$ $= 111111$

Як при шифруванні і розшифруванні, так і при створенні і перевірці підпису, алгоритм RSA у своїй сутності складається з піднесення до степеня, що виконується як ряд множень.

У практичних додатках для відкритого ключа зазвичай обирається відносно невеликий показник, а часто групи користувачів використовують той самий відкритий показник, але кожен з різним модулем. Якщо відкритий показник незмінний, вводяться деякі обмеження на головні співмножники (фактори) модуля. При цьому шифрування даних йде швидше, ніж розшифрування, а перевірка підпису - швидше, ніж підписання.

Якщо *k* - кількість бітів у модулі, то зазвичай в алгоритмах, що використовуються для RSA, кількість кроків, необхідна для виконання операції з відкритим ключем, пропорційна другій степені *k*, кількість кроків для операцій секретного ключа - третій степені *k*, кількість кроків для операції створення ключів - четвертій степені *k*.

Методи «швидкого множення» - наприклад, методи базовані на швидкому перетворенні Фур'є - виконуються меншою кількістю кроків. Проте вони не набули широкого розповсюдження через складність програмного забезпечення, а також тому, що з типовими розмірами ключів вони фактично працюють повільніше. Однак продуктивність та ефективність програм і обладнання, які реалізують алгоритм RSA, швидко збільшується.

Алгоритм RSA набагато повільніший, ніж DES та інші алгоритми блокового шифрування. Програмна реалізація DES працює швидше принаймні у 100 разів і від 1,000 до 10,000 - в апаратній реалізації (залежно від конкретного пристрою). Завдяки проведенню розробок, швидкість алгоритму RSA, ймовірно,

прискориться, але одночасно прискориться і робота алгоритмів блокового шифрування.

Способи злому криптосистеми RSA

Існує кілька способів злому RSA. Найбільш ефективна атака: знайти секретний (*private*) ключ, відповідний необхідному відкритому (*public*) ключу. Це дозволить нападаючому читати всі повідомлення, зашифровані відкритим (*public*) ключем, і підробляти підписи. Таку атаку можна провести, знайшовши головні співмножники (фактори) загального модуля n – p і q. На підставі p, q і e (загальний показник), нападник може легко обчислити приватний показник d. Основна складність – пошук головних множників (факторинг) n; безпека RSA залежить від розкладання на множники (факторингу), що є важким завданням, яке не має ефективних способів вирішення.

Насамперед, завдання відновлення секретного (*private*) ключа еквівалентна задачі розкладання на множники (факторингу) модуля: можна використовувати d для пошуку співмножників n, і навпаки-використовувати n для пошуку d. Треба відзначити, що вдосконалення обчислювального обладнання саме по собі не зменшить стійкість криптосистеми RSA, якщо ключі будуть мати достатню довжину. Фактично ж вдосконалення обладнання збільшує стійкість криптосистеми.

Інший спосіб зламати RSA полягає в тому, щоб знайти метод обчислення кореня степеня e з mod n. Оскільки $c = m^e \bmod(n)$., то коренем степеня e з (mod n) є повідомлення M. Зрозумівши корінь, можна розкрити зашифровані повідомлення та підробляти підписи, навіть не знаючи приватний (private) ключ. Така атака не є еквівалентною факторингу, але в даний час невідомі методи, які дозволяють зламати RSA таким чином. Однак, в особливих випадках, коли на основі одного і того самого показника відносно невеликої величини шифрується досить багато пов'язаних повідомлень, є можливість розкрити повідомлення. Згадані атаки - єдині способи розшифрувати всі повідомлення, зашифровані даними ключем RSA. Проте існують і інші типи атак, що дозволяють, розкрити

тільки одне повідомлення і не дозволяють нападнику розкрити інші повідомлення, зашифровані тим же ключем.

Найпростіший напад на єдине повідомлення – атака по передбачуванню відкритого тексту. Нападник, маючи зашифрований текст, вважає, що повідомлення містить якийсь певний текст, наприклад, «Напад на світанку», потім шифрує передбачуваний текст відкритим (*public*) ключем одержувача і порівнює отриманий текст з наявним зашифрованим текстом. Таку атаку можна запобігти, додавши в кінець повідомлення кілька випадкових бітів. Інша атака єдиного сполучення застосовується в тому випадку, якщо хтось посилає те ж саме повідомлення M трьом кореспондентам, кожен з яких використовує загальний показник $e = 3$. Знаючи це, нападник може перехопити ці повідомлення і розшифрувати повідомлення M.

Таку атаку можна запобігти, вводячи в повідомлення перед кожним шифруванням кілька випадкових біт. Також існують кілька атак щодо зашифрованого тексту (або атаки окремих повідомлень з метою підробки підпису), при яких нападник створює певний зашифрований текст і отримує відповідний відкритий текст, наприклад, змушуючи обманним шляхом зареєстрованого користувача розшифрувати підроблене повідомлення.

Що стосується створення труднощів злому збільшенням розміру ключа, то подвоєння довжини модуля в середньому збільшує час операцій відкритого (*public*) ключа (шифрування і перевірка підпису) в чотири рази, а час операцій секретного (*private*) ключа (розшифровка і підпис)-у вісім разів. Різниця між часом роботи відритого і секретного ключів виникає тому, що відкритий показник може залишатися незмінним, в той час як модуль буде збільшений, а довжина приватного показника буде збільшена пропорційно збільшенню довжини ключа. Час створення ключів при подвоєнні модуля збільшується в 16 разів, але це нечасто виконувана операція і тому на загальній продуктивності це практично не позначається.

Треба відзначити, що розміри ключів у криптосистемі RSA (а також і в інших криптосистемах відкритого (*public*) ключа) набагато більше розмірів

ключів систем блокового шифрування типу DES, але надійність ключа RSA непорівняння з надійністю ключа аналогічної довжини іншої системи шифрування.

Зрозуміло, існують і атаки, націлені нема на криптосистему безпосередньо, а на вразливі місця всієї системи комунікацій в цілому; такі атаки не можуть розглядатися як злом RSA, тому що свідчать не про слабкість алгоритму RSA, а скоріше про уразливість його конкретної реалізації.

Наприклад, нападник може заволодіти секретним (*private*) ключем, якщо той зберігається без належних запобіжних заходів. Необхідно підкреслити, що для повного захисту недостатньо захистити виконання алгоритму RSA і вжити заходів обчислювальної безпеки, тобто використовувати ключ достатньої довжини. На практиці ж найбільший успіх мають атаки на незахищені етапи управління ключами системи RSA.

Особливості алгоритму

Генерація простих чисел

Для знаходження двох великих простих чисел p і q, при генерації ключа, звичайно використовуються ймовірнісні тести чисел на простоту, які дозволяють швидко виявити й відкинути складені числа.

Для генерації p і q необхідно використовувати криптографічно надійний генератор випадкових чисел. У порушника не має бути можливості одержати будь-яку інформацію про значення цих чисел. p і q не повинні бути занадто близькими одне до одного, інакше можна буде знайти їх, використовуючи метод факторизації Ферма. Крім того, необхідно вибирати «сильні» прості числа, щоб не можна було скористатися p-1 алгоритмом Поларда.

Доповнення повідомлень

При практичному використанні необхідно деяким чином доповнювати повідомлення. Відсутність доповнень може призвести до деяких проблем:

– Значення $m = 0$ і $m = 1$ дадуть при шифруванні шифротексти 0 і 1 при будь-яких значеннях e і n.

– При малому значенні відкритого показника ($e = 3$, наприклад) можлива ситуація, коли виявиться, що $m^e < n$. Тоді $c = m^e \bmod n = m^e$, і зловмисник легко зможе відновити вихідне повідомлення, обчисливши корінь степеня e з c.

– Оскільки RSA є детермінованим алгоритмом, тобто не використовує випадкових значень у процесі роботи, то зловмисник може використати атаку з обраним відкритим текстом.

Для розв'язання цих проблем повідомлення доповнюються—перед кожним шифруванням, деяким випадковим значенням. Доповнення виконується таким чином, щоб гарантувати, що $m \neq 0, m \neq 1$ і $m^e > n$. Крім того, оскільки повідомлення доповнюється випадковими даними, то шифруючи той самий відкритий текст, ми кожного разу будемо одержувати інше значення шифротексту, що робить атаку з обраним відкритим текстом неможливою.

Вибір значення відкритого показника

RSA працює значно повільніше симетричних алгоритмів. Для підвищення швидкості шифрування відкритий показник e вибирається невеликим, звичайно 3, 17 або 65537 (2 обрати не можна, бо повинно бути взаємно простим із $j(n) = (p-1)(q-1)$;. Ці числа у двійковому вигляді містять тільки по дві одиниці, що зменшує число необхідних операцій множення при піднесенні до степеня. Наприклад, для піднесення числа m до степеня 17 потрібно виконати тільки 5 операцій множення:

$$m^2 = m * m$$

$$m^4 = m^2 * m^2$$

$$m^8 = m^4 * m^4$$

$$m^{16} = m^8 * m^8$$

$$m^{17} = m^{16} * m$$

Вибір малого значення відкритого показника може призвести до розкриття повідомлення, якщо воно відправляється відразу декільком одержувачам, але ця проблема вирішується за рахунок доповнення повідомлень.

Вибір значення секретного показника

Значення секретного показника d, повинне бути досить великим. У 1990 році Міхаель Вінер показав, що якщо $q < p < 2q$ і $d < n^{\frac{1}{4}}$, то є ефективний спосіб обчислити d по n і e. Однак, якщо значення e вибирається невеликим, то d виявляється досить великим, і проблеми не виникає.

6.2. Індивідуальне завдання

Розробити додаток обміну таємними посиланнями між двома клієнтами за допомогою алгоритму шифрування RSA.

Алгоритм дій:

– Створити ключову пару: приватний та публічний ключі. Ключ(і) повинні бути розташовані поряд з виконуючим файлом.

– Текст повідомлення розташовано в файлі message.txt.

– Клієнт 1 зчитує повідомлення, кодує його за допомогою свого приватного ключа та «відправляє» його клієнту 2 (у найпростішому випадку — розташовує кодований файл поряд з виконуючим файлом клієнта 2). Для підтвердження у подальшому коректності дешифрування, клієнт 1 повідомляє також хеш-суму оригінального повідомлення.

– Клієнт 2 зчитує кодоване повідомлення, дешифрує його за допомогою публічного ключа, знаходить хеш-суму дешифрованого повідомлення, порівнює її з вказаною, та, нарешті, виводить результат про коректністьдешифрування.

Контрольні зпитання

1. Чим відрізняються симетричний та асиметричний типи шифрування?
2. Які переваги має симетричне шифрування?
3. Наведіть приклади алгоритмів симетричного шифрування.
4. Які переваги та недоліки має асиметричне шифрування перед симетричним?

5. Для чого доцільно використовувати асиметричне шифрування?

6. Наведіть приклади алгоритмів асиметричного шифрування.

7. Чим відрізняється відкритий ключ від секретного?

8. З яких етапів складається алгоритм RSA?

9. Які особливості генерації простих чисел необхідно пам'ятати?

10. Які недоліки має алгоритм RSA?

11. Чому слід доповнювати повідомлення при практичному використанні алгоритму RSA?

12. Як слід вибирати значення відкритого показника?

13. Яким чином можна здійснити злом шифрування алгоритмом RSA?

14. Як запобігти атаці по передбачуваному тексту?

15. Як можна створити труднощі щодо злому алгоритму RSA?

ЛАБОРАТОРНА РОБОТА 7

Створення ліцензійного ключа

Мета роботи: дослідити і порівняти існуючі механізми створення і перевірки валідності ліцензійних ключів.

7.1 Теоретичні відомості

Ліцензійний ключ – це специфічний програмний ключ для комп'ютерної програми. Він засвідчує, що копія програми є оригіналом.

Ключі складаються з ряду цифр та/або букв. Цю послідовність, як правило, вводить користувач під час установлення комп'ютерного програмного забезпечення, а потім передає функції перевірки до програми. Ця функція маніпулює послідовністю ключів відповідно до математичного алгоритму та намагається зіставити результати з набором допустимих рішень.

Генерація стандартних ключів, коли ключі продукту генеруються математично, не є повністю ефективною для припинення порушення авторських прав програмного забезпечення, оскільки ці ключі можна розповсюджувати. Крім того, завдяки покращенню комунікації з розвитком Інтернету, стали досить поширеними більш складні атаки на ключі, такі як crack (усунення потреби в ключі) та генератори ключів продукту.

Через це видавці програмного забезпечення використовують додаткові методи активації продуктів, щоб перевірити, чи є ключі дійсними та безкомпромісними. Один із методів присвоює ліцензійний ключ на основі унікальної особливості апаратного забезпечення покупця, яку неможливо так легко продублювати, оскільки це залежить від апаратного забезпечення користувача. Інший метод передбачає необхідність одноразової або періодичної перевірки ключа продукту за допомогою Інтернет-сервера (для ігор з онлайн-компонентом це робиться щоразу, коли користувач входить у систему). Сервер може деактивувати немодифіковане клієнтське програмне забезпечення,

представляючи недійсні або порушені ключі. Модифіковані клієнти можуть обійти такі перевірки, але сервер все одно може заборонити цим клієнтам інформацію чи зв'язок.

Деякі з найефективніших засобів захисту продукту суперечливі через незручності, суворі штрафи та, в деяких випадках, помилкові спрацьовування. Деякі ключі продукту використовують незручні та суворі цифрові процедури для забезпечення ліцензійної угоди.

Недоліки

Ліцензійні ключи дещо незручні для кінцевих користувачів. Їх потрібно не тільки вводити щоразу, коли встановлюється програма, але й користувач повинен бути впевнений, що не втратить їх. Втрата ключа продукту, як правило, означає, що після видалення програмне забезпечення марне, якщо перед видаленням не використовується програма відновлення ключа (хоча не всі програми це підтримують).

Існує багато випадків блокувань, що застосовуються компаніями, при виявленні порушень користування. Інтернет-система зазвичай негайно вносить у чорний список обліковий запис, на якому запущені заборонені програми (такі як crack). Це призводить до постійного блокування.

Особливо суперечливою є ситуація, яка виникає, коли ключі кількох продуктів пов'язані між собою. Якщо товари мають залежність від інших продуктів (як це має місце з пакетами розширень), зазвичай компанія забороняє всі пов'язані товари. Наприклад, якщо підроблений ключ використовується з пакетом розширення, сервер може заборонити законні ключі від оригінального програмного забезпечення.

Об’ємне ліцензування

При ліцензуванні програмного забезпечення об'ємне ліцензування – це практика продажу ліцензії, яка дозволяє одній комп'ютерній програмі користуватися великою кількістю комп'ютерів або великою кількістю користувачів. Замовниками таких схем ліцензування є, як правило, бізнес,

державні чи освітні установи, при цьому ціни на обсяжне ліцензування варіюються залежно від типу, кількості та застосовного терміну підписки. Наприклад, програмне забезпечення Майкрософт, доступне за допомогою програм корпоративного ліцензування, включає Microsoft Windows та Microsoft Office.

Зазвичай ключ об'ємного ліцензування (Volume License Key), який міг бути наданий у всі екземпляри ліцензованої комп'ютерної програми, був задіяний в об’ємному ліцензуванні. З огляду на популярність програмного забезпечення як практики обслуговування, клієнти корпоративного ліцензування постачають лише програмне забезпечення з обліковими даними, що належать обліковому запису Інтернету, який використовується для інших аспектів надання послуг.

Традиційно ключ продукту постачається разом із комп’ютерними програмами. Він діє аналогічно паролю: комп’ютерні програми старих користувачей просять підтвердити своє право; у відповідь користувач надає цей ключ. Однак цей ключ слід використовувати лише один раз, тобто на одному комп'ютері. Однак ключ кількісного ліцензування (VLK) можна використовувати на кількох комп’ютерах. Постачальники можуть вжити додаткових заходів, щоб гарантувати, що ключ від їхньої продукції використовується лише у призначеному для цього номері. Ці зусилля називаються активацією продукту.

Об’ємні ліцензії не завжди передаються. Наприклад, можна передавати лише деякі типи корпоративної ліцензії Microsoft за умови завершення формального процесу передачі, який дозволяє корпорації Майкрософт зареєструвати нового власника. Дуже невелика кількість постачальників програмного забезпечення спеціалізується на посередництві таких передач, щоб дозволити продаж кількісних ліцензій та ключів.

Процес ліцензування програмного забезпечення

На сьогодні будь-яка покупка програмного забезпечення починається з того, що користувач вносить свою клієнтську інформацію в базу даних сервера, який володіє ліцензією на дане програмне забезпечення. За допомогою надбудов

сервісів програмного забезпечення з обробки ліцензій користувач передає свої клієнтські дані, які збережуться на сервері і будуть відображати його в клієнтській базі.

До таких даних на поточний момент належать:

– E-mail користувача;

– ім'я компанії, яку представляє даний користувач. У разі, якщо користувач — фізична особа, це поле може залишатися порожнім;

– ПІБ користувача, на ім'я якого відбувається реєстрація;

– пароль користувача.

При цьому, e-mail і пароль користувача можуть виступати в процесі аутентифікації користувача, у зв'язку з цим e-mail повинен бути унікальний в системі, а пароль — бути безпечним згідно з політикою паролів.

Після того, як система-сервер отримала інформацію про користувача, активізуються такі дії:

– Перевіряється наявність даного користувача у своїй базі даних. У разі, якщо користувач з таким ідентифікатором-email вже існує, повертається відповідь з відповідною помилкою.

– Генерується пара ключів, з урахуванням власного центра сертифікації, кінцевого користувача для можливості подальшого безпечного обміну повідомлення з ним.

– Зберігається в базі даних передана інформація про користувача, а також пара ключів. При цьому, в цілях захисту призначених для користувача даних, пароль користувача не зберігається у відкритому вигляді, а зберігається тільки його хеш-сума.

– Як відповідь, сервер повертає клієнтові публічний ключ сервера і приватний ключ клієнта для можливості безпечного обміну повідомленнями. Дана відповідь не підлягає додатковим засобом захисту від зловмисника.

Отримавши успішну відповідь від сервера, клієнт зберігає отримані ключі, а також свій e-mail у клієнтській базі даних.

Далі за допомогою технічних систем ідентифікації та ліцензування ПО формується ліцензійний ключ. Відповідно до передбачуваних процедур формування ліцензійного цифрового ідентифікатора необхідно виконати такі кроки:

1. Формування шаблону програмного коду, на основі якого генерується ліцензійний ключ.
2. Отримання інформації про компоненти системи кінцевого користувача. Ця функція виконується шляхом створення програми-утиліти, яка запускається на системі кінцевого користувача і відправляє по захищеному каналу необхідну інформацію на сервер, щоб уникнути перехоплення трафіку і його аналізу/зміни. Отримана інформація буде зберігатися для подальших дій.
3. Формування результуючого ліцензійного ключа. Для виконання даної функції як вхідних даних використовуються: інформація про системи кінцевого користувача; шаблон програмного коду, який буде виконуватися; розмір CRC суми для валідації ліцензійного ключа.

Узагальнений алгоритм процесу формування цифрового ідентифікатора ПО:

1. Формування програмного коду (на асемблері), який буде виконуватися на системі кінцевого користувача з урахуванням отриманої інформації і наданого шаблону.
2. Отриманий код трансформується у відповідні машинні коди.
3. Для кожного байта отриманого машинного коду виконуються такі дії:
   - пошук даного байта в текстовому поданні отриманої інформації;
   - пошук даного байта в бінарному подання отриманої інформації;
   - якщо знайдений відповідний код у текстовому поданні, то в кінець формованого ліцензійного ключа дописується 2 байта: «0» і «код пристрою », якщо в бінарному поданні — то 2 байта: «1» і «код пристрою», потім 2 байта, що

відображають позицію знайденого символу. У разі якщо код не був знайдений — 2 байта «00», а потім 2 байта, що відображають hex уявлення зазначеного байта (наприклад, символ «A» має код «65», що в hex-поданні — 41 - буде записано два байта: «4» і «1»).

У кінець отриманого ліцензійного ключа додається 2 байта, що характеризують ступінь CRC суми, наприклад «32». Далі йде CRC сума в текстовому hex-поданні (для CRC32 кількість – 8 символів).

Початковий програмний продукт проходить 3 фази перед надходженням до кінцевого користувача:

1. Отриманий ліцензійний ключ вбудовується в сегмент даних програми. При цьому сегмент даних повинен бути сформований таким чином, щоб ліцензійний ключ помістився у виділеному для цього блоці.

2. Обфускація — для зменшення ймовірності злому алгоритму обробки ліцензійного ключа.

3. Підпис програмного продукту сертифікатом — для уникнення можливості редагування файлу з метою обійти алгоритм верифікації.

Далі отриманий програмний комплекс поставляється кінцевому користувачеві. При кожному запуску програма виконує такі дії:

1. Отримує інформацію про компоненти поточної системи.
2. Верифікація ліцензійного ключа на основі CRC суми.
3. Формування машинних кодів для виконання на основі поточного ліцензійного ключа і параметрах системи.
4. Виконання сформованих машинних команд. У разі якщо ліцензійний ключ виявився невалідним або програма була запущена на іншому комп'ютері, то поточний програмний продукт аварійно завершить роботу.

Генератори ключів

Генератор ключів (key-gen) – це комп'ютерна програма, яка генерує ключ ліцензування продукту, такий як серійний номер, необхідний для активації при використанні програмного додатка. Кейгени можуть законно розповсюджуватися виробниками програмного забезпечення для ліцензування програмного забезпечення в комерційних середовищах, де програмне забезпечення було ліцензовано масово для всього веб-сайту або підприємства, або вони можуть розповсюджуватися незаконно за обставин порушення авторських прав або піратства програмного забезпечення. Генератори незаконних ключів, як правило, поширюються програмними зломниками.

Окремо слід роздивитися погрози злому ліцензії.

Команди, які спеціалізуються на зломі ПЗ, пропонують їх на різних сайтах, присвячених поширенню ПЗ без дотримання ліцензії. Оскільки ліцензії більшості ПЗ, що використовують ключі активації, вимагають обов'язкової наявності ліцензійних кодів, використання кейгенів для неоплаченого власницького ПЗ зазвичай є незаконним. Багато програм задля додаткової безпеки перевіряють ліцензійні ключі через Інтернет шляхом встановлення сеансу за допомогою програми ліцензування видавця програмного забезпечення. Розширені кейгени обходять цей механізм і включають додаткові функції для перевірки ключів, наприклад, шляхом генерації даних перевірки, які в іншому випадку повертаються сервером активації. Якщо програмне забезпечення пропонує телефонну активацію, то кейген може сформувати правильний код активації для її завершення. Іншим методом, який був використаний, є емуляція сервера активації, яка виправляє пам'ять програми, щоб «побачити» кейген як фактичний сервер активації.

7.2. Індивідуальне завдання

1. Дослідити існуючі механізми створення і перевірки валідності ліцензійних ключів. Зробити порівняльну характеристику кожного механізму.
2. Механізм генерації ліцензійного ключа №1.
   - Створити пару ключів (приватний та публічний).
   - Створити додаток, що генерує ліцензійний ключ, який має інформацію про кінцевого користувача. У кінці цього ЛК повинна бути ЕЦП – хеш-сума, для калькуляції якої використовувався приватний ключ.
   - Створити додаток, що читає ліцензійний ключ та виводить його дані на екран (у тому числі, чи валідний «підпис», використовуючи публічний ключ).
3. Механізм генерації ліцензійного ключа №2.
   - Створити клієнт-серверний додаток. Задача веб-сервера : отримати рядок-ключ та повернути відповідь, чи валідний цей ключ (перелік ключів зберігається на сервері, метод зберігання не має значення). Задача клієнта: Спитати у користувача ключ, отримати відповідь, на базі якої вивести, чи має змогу користувач далі користатися даним ПЗ.

Зверніть увагу. Результатом виконання цієї роботи є зібрані jar файли та bat/shell файл для запуска. Не є дозволеним завантажувати додаток засобами IDE!

Контрольні запитання

1. Що таке ліцензійний ключ?
2. Які існують додаткові методи активації продукту?
3. Які недоліки має використання ліцензійних ключів?
4. Що таке об’ємне ліцензування?
5. Чим ключ об’ємного ліцензування відрізняється від звичайного?
6. Які дані належать до клієнтської інформації?

7. Як система сервер реагує на отримання клієнтської інформації?

8. Які кроки необхідно виконати для створення ліцензійного ключа?

9. Як виглядає узагальнений алгоритм процесу формування цифрового ідентифікатора ПЗ?

10. Які фази початковий програмний продукт проходить перед надходженням до кінцевого користувача?

11. Які дії ліцензована програма має виконувати при кожному запуску?

12. На основі якого параметра проходить верифікація ліцензійного ключа?

13. Що таке генератор ключів?

14. Як розробники можуть попередити злом ліцензії програмного забезпечення?

15. Як зловмисники обходять механізм ліцензування програмного забезпечення?

ЛАБОРАТОРНА РОБОТА 8

***Java*-агенти**

Мета роботи: дослідити можливості java-агентів.

8.1. Теоретичні відомості

Java-агент – це звичайний клас Java, який підкоряється суворим правилам. Клас агента повинен реалізовувати *public static void premain (String agentArgs, Instrumentation inst),* який стає точкою входу агента (аналогічно методу main для звичайних додатків Java).

Після ініціалізації віртуальної машини Java (JVM) кожен такий *premain* метод *(String agentArgs, Instrumentation inst)* кожного агента буде викликатися в тому порядку, в якому агенти були вказані при запуску JVM. Коли цей крок ініціалізації буде виконаний, буде викликаний main метод реального додатка Java.

Однак, якщо клас не реалізує *public static void premain (String agentArgs, Instrumentation inst)*, JVM спробує знайти і викликати іншу перевантажену версію *public static void premain (String agentArgs)*. Будь ласка, зверніть увагу, що кожен метод premain повинен повертатися для продовження фази запуску.

Нарешті, що не менш важливо, клас агента Java також може мати *public static void agentmain (String agentArgs, Instrumentation inst)* або *public static void agentmain (String agentArgs)* методи, які використовуються при запуску агента після запуску JVM. Наступні атрибути визначені для агентів Java, які упаковані як файли архіву Java (або просто файли JAR).

Таблиця 8.1 – Атрибути

Атрибут маніфесту	Опис
Premain-Class	Коли агент вказується під час запуску JVM, цей атрибут визначає клас агента

	Java: клас, який містить метод *premain*. Якщо під час запуску JVM вказано агента, цей атрибут обов'язковий. Якщо атрибут відсутній, JVM буде перервана.
Agent-Class	Якщо реалізація підтримує механізм запуску агентів Java через деякий час після запуску JVM, тоді цей атрибут вказує клас агента: клас, який містить метод *agentmain*. Цей атрибут є обов'язковим, і агент не буде запущений, якщо цей атрибут відсутній.
Boot-Class-Path	Список шляхів для пошуку завантажувачем класів початкового завантаження. Шляхи подають каталоги або бібліотеки.
Can-Redefine-Classes	Значення true або false, не залежить від регістра і визначає можливість перевизначення класів, необхідних цього агента. Цей атрибут є необов'язковим, за замовчуванням використовується значення false.
Can-Retransform-Classes	Значення true або false, невідчутно до регістра і визначає, чи потрібна цьому агентові можливість повторного перетворення класів. Цей атрибут є необов'язковим, за замовчуванням використовується значення false.

Can-Set-Native-Method-Prefix	Значення true або false, невідчутне до регістра і визначає, чи потрібна цьому агентові можливість встановлювати власний префікс методу. Цей атрибут є необов'язковим, за замовчуванням використовується значення false.

Java-агент і інструментарій

До інструментальних можливостей агентів Java належать:

– Можливість перевизначити класи під час виконання. Перевизначення може змінити тіла методу, пул констант і атрибути. Перевизначення не повинно додавати, видаляти або перейменовувати поля або методи, змінювати сигнатури методів або змінювати успадкування.

– Можливість повторного перетворення класів під час виконання. Ретрансформація може змінити тіла методу, пул констант і атрибути. Повторне перетворення не повинно додавати, видаляти або перейменовувати поля або методи, змінювати сигнатури методів або змінювати успадкування.

– Можливість змінити обробку помилок дозволу власних методів, дозволивши повторну спробу з префіксом, застосованим до імені.

Зверніть увагу, що повторно перетворений або перевизначений байт-код класу не перевіряється і не встановлюється відразу після застосування перетворень або перевизначень. Якщо виходить ,що в результаті байт-код помилковий або невірний, буде згенеровано виняток, який може повністю викликати збій JVM.

Метод *premain* може мати одну з таких сигнатур:

public static void premain(String agentArgs, Instrumentation inst);

public static void premain(String agentArgs); (тільки Java 1.6).

У *agentArgs* передається сам рядок options, розробник повинен сам реалізувати логіку парсинга цього рядка, якщо хоче передавати в ньому кілька аргументів. Власне сам інтерфейс *java.lang.instrument.Instrumentation* і надає

доступ до механізмів операцій з байт-кодом. Сам метод premain викликається ще до виконання методу main.

```
public class InstrumentExample {
   private static Instrumentation inst;
   public static void premain(String options, Instrumentation inst) {
      inst.addTransformer(new SomeTransformer());
      this.inst = inst;
   }
   public static void main(String args[] ) {
      ...
      byte[] classBytes = ...
      ClassDefinition classDef = new ClassDefinition(SomeClass.class,
classBytes);
      inst.redefineClasses(classDef);
      ...
   }
   public static class SomeTransformer implements ClassFileTransformer {
      public byte[] transform(java.lang.ClassLoader loader,
            java.lang.String className,
            java.lang.Class classBeingRedefined,
            java.security.ProtectionDomain protectionDomain,
            byte[] classfileBuffer) throws IllegalClassFormatException
      {
         ...
      }
   }
   ...
}
```

Щоб використовувати механізм трансформації класів, вам потрібно реалізувати інтерфейс *java.lang.instrument.ClassFileTransformer.* і зареєструвати свою реалізацію через метод *Instrumentation.addTransformer*. Сам трансформер буде спрацьовувати кожного разу при:

– завантаженні класу *ClassLoader.defineClass;*

– перевизначенні класу *Instrumentation.redefineClasses;*

– ретрансформації класу *Instrumentation.retransformClasses.*

Аргумент *classfileBuffer* містить байт-код поточної версії класу, і його не можна модифікувати, для його перевизначення трансформер повинен повернути новий масив байтів, або null, якщо трансформація не була проведена. У разі, коли трансформерів кілька, то вони будуть викликані по ланцюжку, при цьому *classfileBuffer* буде результатом попереднього трансформера. Щоб повідомити про помилку при трансформації, необхідно викинути *java.lang.instrument.IllegalClassFormatException*, при *unchecked*-помилках результат буде такий, як якщо б метод повернув *null*.

Для перевизначення класу необхідно вказати вище описаний параметр у маніфесті і використовувати клас *java.lang.instrument.ClassDefinition*. Проте, перевизначення класу не повинно додавати, видаляти нові поля або методи, змінювати сигнатуру методів або ієрархію спадкування. Можна змінювати тільки тіло методів, структура класу змінюватися не повинна. Якщо в додатку вже використовуються екземпляри попередньої версії класу, то з ними нічого не станеться, однак при подальшому створенні класу буде використана вже нова версія.

Приклад

Інструменти Profiler часто повідомляють про різні параметри об'єктів Java під час виконання, витягуючи інформацію з JVM. Ці параметри включають інформацію про використання пам’яті об’єкта тощо, використовуючи інструментальну структуру.

Це зразок класу, який ми використаємо в нашому прикладі:

```
package com.mano.examples;
public class Main {
  public static void greet(String msg){
    System.out.println(msg);
  }
  public static void main(String[] args){
    greet("Hello Agents");
  }
}
```

Клас інструментального агента з методом premain використовується для отримання необхідної інформації. Реалізація інтерфейсу Instrumentation передається методу premain. Ми використовуємо метод getObjectSize, визначений інтерфейсом контрольно-вимірювальних приладів, щоб отримати використання пам'яті основного об'єкта під час виконання:

```
package com.mano.examples;
import java.lang.instrument.Instrumentation;
public class MyAgentClass {
  public static void premain(String agentArgs,
    Instrumentation inst) {
    System.out.println(inst.getObjectSize
      (new Main()))
  }
}
```

Після цього ми повинні створити файл MANIFEST.MF. Це не що інше, як текстовий файл, куди ми поміщаємо інформацію, пов’язану з класом агента. JVM

використовуватиме його для завантаження агента. Файл зазвичай зберігається в каталозі META-INF. Вміст, необхідний для нашого прикладу, є досить простим:

Manifest-Version: 1.0

Premain-Class: com.mano.examples.MyAgentClass

Тепер скомпілюйте всі файли Java, щоб створити файл класу. І нарешті, створіть файли JAR наступним чином:

jar -cmf META-INF/MANIFEST.MF myagent.jar com/mano/examples/ MyAgentClass.class

Після створення агента він розгортається як файл JAR. Атрибут у файлі маніфесту визначає клас агента, який буде завантажений для запуску агента. Зверніть увагу, що існує багато способів запустити агент: за допомогою командного рядка, під час виконання або як виконувані файли JAR. Тут ми використаємо командний рядок.

Запуск агента за допомогою командного рядка такий:

Командний рядок:

java -javaagent:myagent.jar -cp . com.mano.examples.Main

Це вказує на те, що метод *premain* буде запущений до виконання програми і буде створено розмір екземпляра *Main*.

8.2. Індивідуальне завдання

Зверніть увагу, що для виконання цієї роботи вам буде потрібен проєкт попередньої роботи (Ліцензійні ключі). Треба використовувати виконуючі файли цієї роботи (без модифікації коду).

Пре-реквізити:

– Створити нову пару ключів.

– Створити ліцензійний ключ для першого проєкту, що підписаний новим секретом.

– Підтвердити, що «клієнт» вважає ключ невалідним, тому що він некоректно підписаний.

Завдання:

Використовуючи механізм java-агентів, виконати такі дії:

– «підключитися» до додатка "клієнта" до етапу завантаження класів;

– вивести статистику завантажених класів: завантажених класів самого додатка, завантажених класів jre, завантажених класів використаних зовнішніх бібліотек (якщо такі є);

– виконати підміну публічного ключа (не модифікуючи сам файл ключа).

Після виконання описаних кроків вдосконалитись, що ліцензійний ключ, створений у розділі «пре-реквізити», є тепер валідним.

Контрольні запитання

1. Що таке Java-агент?
2. Чим Java-агент відрізняється від звичайного класу?
3. Назвіть умову продовження фази запуску java-агента.
4. Якої умови потрібно дотриматися для того, щоб використовувати механізм трансформації класів?
5. Які методи може мати клас java-агента?
6. Назвіть атрибути java-агента.
7. Які існують особливості перевизначення класу?
8. Які особливості виклику має аргумент *classfileBuffer*?
9. Назвіть інструментальні можливості агентів Java.
10. Скільки аргументів має метод *premain*?
11. Що таке Manifest-файл?
12. Як виглядає java-агент після створення?

ЛАБОРАТОРНА РОБОТА 9

Підміна видаленого серверу

Мета роботи: дослідити особливості підміни сервера.

9.1. Теоретичні відомості

Підміна IP-адреси

Основним протоколом надсилання даних через мережу Інтернет та багато інших комп'ютерних мереж є Інтернет-протокол (IP). Протокол визначає, що кожен пакет IP повинен мати заголовок, який містить (серед іншого) IP-адресу відправника пакета. Структуру заголовка пакета IP наведено в табл. 9.1.

Таблиця 9.1 – Структура заголовка пакета IP

<table>
<tr><td>Номер версії</td><td>Довжина заголовка</td><td>Тип сервісу</td><td>Загальна довжина</td></tr>
<tr><td colspan="2">Ідентифікатор «великого пакета»</td><td>Прапори</td><td>Зсув фрагмента</td></tr>
<tr><td>Час життя</td><td>Протокол</td><td colspan="2">Контрольна сума заголовка</td></tr>
<tr><td colspan="4">Адреса відправника</td></tr>
<tr><td colspan="4">Адреса одержувача</td></tr>
<tr><td>Опції</td><td colspan="3">Вирівнювання</td></tr>
</table>

Зазвичай вихідною IP-адресою є адреса, з якої був відправлений пакет, але адресу відправника в заголовку можна змінити так, що одержувачу здається, нібито пакет надійшов з іншого джерела.

Протокол вимагає від комп'ютера-одержувача відповіді на вихідну IP-адресу, так що підробка використовується в основному, коли відправник може передбачити мережеву відповідь або не піклується про відповідь.

IP-адреса джерела надає лише обмежену інформацію про відправника. Він може надати загальну інформацію про регіон та місто, з якого надіслано пакет. Він не надає інформації про особу відправника або використовуваний комп’ютер.

Для зловмисника базовий принцип атаки полягає у фальсифікації власних заголовків IP-пакетів, в яких змінюється, серед іншого, IP-адреса джерела. Атака IP-spoofing часто називається «сліпою підміною» (Blind Spoofing). Це пов'язано з тим, що відповіді на фальсифіковані пакети не можуть прийти машині зловмисників, тому що був змінено вихідну адресу. Однак все-таки існують два методи отримання відповідей:

– Маршрутизація від джерела (en: Source routing): в протоколі IP існує можливість маршрутизації від джерела, яка дозволяє задавати маршрут для відповідних пакетів. Цей маршрут є набором IP-адрес маршрутизаторів, через які повинен пройти пакет. Для крекерів досить надати маршрут для пакетів до маршрутизатора, ним контрольованого. У наш час більшість реалізацій стека протоколів TCP / IP відбраковують пакети з маршрутизацією від джерела.

– Перемаршрутизація (Re-routing): якщо маршрутизатор використовує протокол RIP, то його таблиці можна змінити, надсилаючи йому RIP-пакети з новою інформацією про маршрути. За допомогою цього зловмисник намагається здійснити перенапрямування пакетів на підконтрольний йому маршрутизатор.

Імітація IP-адреси

Для Unix-систем є можливою атака з імітацією ip-адреси. При імітації IP-адреси комп'ютерна система вважає, що вона взаємодіє з іншою системою. При налаштуванні служби rlogin або rsh IP-адреса відправника є важливим компонентом, оскільки він визначає, кому дозволено використовувати ці служби. Удалені хости, допущені до таких сполук, називаються довіреними. При використанні імітації IP-адреси довіреної системи можна успішно використати цільову систему.

При виявленні системних, довідкових відносин з іншою системою та що знаходяться в мережах, до яких можна підключитися, виконується імітація IP-

адреси, що дозволить хакеру отримати доступ до цієї системи. Однак є ще одна проблема, яку потрібно вирішити. Цільова система у відповідь на додаткові пакети буде надсилати дані довіреної системи. Відповідно до специфікації довіреної системи TCP може відповісти перезавантаженням або пакетом брошури (RST-пакет), оскільки вона не має даних про підключення. Хакер не повинен допускати цього та зазвичай виконує DoS-атаку проти довіреної системи.

При підключенні до служби rlogin хакер може зареєструватися як користувач довіреною системою (неплохим варіантом буде вхідний запис у root, що має ім'я всіх Unix-систем). Якщо він забезпечить себе більш зручним способом входу в систему (при підключенні за допомогою імітації IP-адреси хакер не отримує відповіді цільової системи на свої дії.), він зможе налаштувати цільову систему для дозволу доступу за допомогою логіну з віддаленою системою або додати запит для власного використання.

Підміна ARP

Протокол ARP (Протокол дозволів адрес, Протокол розрішення адрес) реалізує механізм дозволу IP-адреси в MAC-адресі Ethernet. Мережне обладнання встановлює між собою зв'язок за допомогою обміну Ethernet-фреймів (естетично в Ethernet-мережах), на канальному рівні. Для забезпечення можливості передачі цієї інформації необхідно, щоб кожен мережевий інтерфейс мав свою унікальну адресу в мережі Ethernet. Вона називається MAC-адресою (MAC - Media Access Control).

При відправленні IP-пакетів слід знати MAC-адресу отримувача. Щоб її дізнатись, у локальній мережі поширюється широкомовний ARP-запит. У ній запитується «Яка MAC-адреса відповідає певній IP-адресі?». Машина з відповідною IP-адресою відповідає ARP-пакету, що містить потрібну MAC-адресу. З цього моменту відправляюча машина знає MAC-адресу, що відповідає призначеній IP-адресі. Ця відповідність забезпечує збереження деякого часу в кеші (щоб не виконати запрошення кожного разу при посиланні IP-пакетів).

Дана атака змінює кеш цільової машини. Зловмисник відсилає ARP-відповіді цільової машини з інформацією про нову MAC-адресу, відповідній IP-адресі шлюзу. Насправді ця MAC-адреса відповідає інтерфейсу машин зловмисників. Далі весь трафік до шлюзу буде отримувати машину зловмисника. Тепер можна прослухати трафік (і/або змінювати його). Після цього трафік буде створено для реальної цільової адреси і таким чином нікто не помістити зміни.

Атака ARP-spoofing використовується в локальній мережі, побудованій на комутаторах. З її допомогою можна переправити потік ethernet-фреймів на інші порти, відповідно до MAC-адреси. Після чого зловмисник може перехопити всі пакети на своєму порту. Таким чином, атака ARP-spoofing дозволяє перехоплювати трафік машин, розташованих на різних порталах комутатора. Для реалізації атаки ARP-spoofing зловмисник може використовувати генератори ARP-пакетів

Безпека DNS-сервера

DNS (Domain Name System – система доменних імен) – комп’ютерна розподілена система для одержання інформації про домени. Найчастіше використовується для одержання IP-адреси за іменем хоста (комп’ютера або пристрою). Розподілена база даних DNS підтримується за допомогою ієрархії DNS-серверів, що взаємодіють за певним протоколом. Система DNS містить ієрархію DNS-серверів, що відповідає ієрархії зон. Кожна зона підтримується як мінімум одним авторитетним сервером DNS, на якому розташована інформація про домен. Ім’я й IP-адреса не тотожні – одна IP-адреса може мати безліч імен, що дозволяє підтримувати на одному комп’ютері безліч веб-сайтів (це називається віртуальний хостинг). Зворотне теж справедливе – одне ім’я може бути зіставлене з безліччю IP-адрес: це дозволяє створювати балансування навантаження. Для підвищення стійкості системи використовується безліч серверів, що містять ідентичну інформацію, а в протоколі є засоби, що дозволяють підтримувати синхронність інформації, розташованої на різних серверах.

DNS має такі характеристики:

– Розподіленість адміністрування. Відповідальність за різні частини ієрархічної структури несуть різні люди або організації.

– Розподіленість зберігання інформації. Кожен вузол мережі в обов'язковому порядку повинен зберігати тільки ті дані, які входять в його зону відповідальності, і (можливо) адреси кореневих DNS-серверів.

– Кешування інформації. Вузол може зберігати деяку кількість даних не зі своєї зони відповідальності для зменшення навантаження на мережу.

– Ієрархічна структура, в якій всі вузли об'єднані в дерево, і кожен вузол може або самостійно визначати роботу нижчих вузлів, або делегувати (передавати) їх іншим вузлам.

– Резервування. За зберігання та обслуговування своїх вузлів (зон) відповідають (зазвичай) декілька серверів, розділених як фізично, так і логічно, що забезпечує збереження даних і продовження роботи навіть у разі збою одного з вузлів.

Атаки на DNS-сервер

DNS-протокол може працювати як поверх TCP (TransmissionControlProtocol), так і поверх UDP (UserDatagramProtocol), причому в 99 % випадків використовується саме UDP – як більш швидкий, менш ресурсномісткий, але в той же час і менш захищений. Щоб послати підроблений пакет, який буде сприйнятий жертвою як правильний, досить вгадати (підібрати) ідентифікатор послідовності і номер порту-відправника. У найпростішому випадку зловмисник може відправити підроблену DNS-відповідь з підробленою IP-адресою деякого вузла, на який жертва намагається зайти. Складність реалізації атак подібного роду в тому, що робочі станції кешують DNS-запити. Більш того, система не приймає DNS-відповідей, що не запрошувалися. Хакер повинен дочекатися моменту, коли жертва відправить DNS-запит, і згенерувати підроблену відповідь перш, ніж це зробить справжній DNS-сервер. Насправді, обидві проблеми мають рішення. DNS-кеш зазвичай невеликий, а тому, пославши жертві HTML-лист з купою картинок, що лежать на зовнішніх серверах з різними доменними іменами, хакер може витіснити з кеша всі старі записи. Після чого,

остання посилання в листі, що ведуть на сервер оновлень, гарантовано відправить позначений запит в Мережу. Попереднє його посилання на Web-сервер, підконтрольне хакеру, підкаже точний час, коли слід починати генерацію підроблених пакетів. Якщо хоча б один з них буде сприйнятий як правильний, в DNS-кеш потрапить «ліва» адреса сервера з оновленнями, що має всі шанси «дожити» до чергової сесії оновлень.

Атаки на DNS можна умовно розділити на два види:

– Пасивні – атакуючий отримує необхідну інформацію без помітного впливу на систему; система при цьому продовжує функціонувати як перш.

– Активні – атакуючий реалізує деякий вплив на систему, в результаті якого змінюється її поведінка. Така зміна може бути і невизначеною для атакуючої системи, але криптоаналитик у змозі визначити і використовувати цю інформацію.

Однією з основних атак є DNS-hijacking. Перехоплення DNS – підрив дозволу DNS-запитів. Це може бути досягнуто за допомогою шкідливих програм, які перекривають TCP/IP-конфігурації комп'ютера, щоб запити відправлялися на DNS-сервер зловмисника, або через зміну поведінки довіреної DNS-сервера так, щоб воно не відповідало стандартам Інтернету. Ці зміни можуть бути зроблені в зловмисних цілях, таких як фішинг, або в корисливих цілях Інтернет-провайдерами (ISP), направляючими веб-трафік користувача на власні веб-сервери для показу реклами, збору статистики або інших цілей провайдера; і постачальниками послуг DNS для блокування доступу до обраних доменів як форма цензури.

Інший різновид атаки, DNS-spoofing, також відомий як отруєння кеша DNS, є однією з форм злому комп'ютерних мереж, в якому дані кеша доменних імен змінюються зловмисником, з метою повернення помилкової IP-адреси. Це призводить до атаки посередника на комп'ютер зловмисника (або будь-який інший комп'ютер). Як правило, мережевий комп'ютер використовує DNS-сервери, що надаються Інтернет-провайдером (ISP), або власний сервер організації. DNS-сервери використовуються в організації мережі, щоб поліпшити продуктивність

за рахунок кешування раніше отриманих результатів. Отруєння кешу на одному DNS-сервері може вплинути на користувачів, що обслуговуються безпосередньо на віддаленому сервері, або на тих, хто обслуговується даним сервером.

Щоб виконати отруєння кешу, зловмисник експлуатує недоліки програмного забезпечення сервера DNS. Сервер повинен правильно перевірити DNS відповіді, щоб переконатися, що вони належать до надійного джерела (наприклад, за допомогою DNSSEC); в іншому випадку сервер може кешувати невірні записи локально і надавати їх іншим користувачам, які здійснюють запит.

Ця атака може бути використана для переадресації користувачів з сайту на інший сайт за вибором зловмисника. Наприклад, зловмисник підміняє IP-адреса цільового веб-сайту на даному DNS-сервері і замінює її на IP-адресу сервера під власним контролем. Зловмисник створює файли на власному сервері з іменами, що збігаються з тими, що знаходяться на цільовому сервері. Ці файли зазвичай містять шкідливий контент, такий як комп'ютерні хробаки або віруси. Користувач, чий комп'ютер посилається на отруєні DNS-сервери обманом, отримує контент, що надходить від неавторизованого сервера і неусвідомлено завантажує шкідливий контент. Цей метод може також використовуватися для фішингових атак, де фальшива версія справжнього сайту створюється для того, щоб зібрати особисті дані, такі як номери банківських і кредитних/дебетових карт.

9.2. Індивідуальне завдання

Використовуючи другий проєкт роботи з ліцензійними ключами:

– створити додаток, «що прийматиме посилання», адресовані до оригінального сервера та повертає завжди позитивну відповідь;

– зробити підміну а) dns-адреси, б) ip-адреси таким чином, щоб інформація про верифікацію ліцензійного ключа надходила не на валідний сервер, а на підроблений.

Запропонувати варіанти уникнення цих вразливостей.

Контрольні запитання

1. Яку структуру має заголовок IP-пакета?
2. В чому полягає суть атаки IP-spoofing?
3. Назвіть методи отримання відповідей при «сліпій підміні».
4. Яку проблему потрібно контролювати при імітації IP-адреси?
5. Як працює протокол ARP?
6. Як здійснюється ARP-spoofing?
7. В яких мережах зазвичай використовують ARP-spoofing?
8. Що таке DNS?
9. Які характеристики має DNS?
10. Які умови потрібно виконати для підміни DNS?
11. На які види можна поділити DNS-атаки?
12. В чому суть атаки DNS-hijacking?
13. На чому базована атака DNS-spoofing?
14. Як виконується DNS-spoofing?
15. Для чого зазвичай використовується DNS-spoofing?

ЛАБОРАТОРНА РОБОТА 10

Оцінка якості коду

Мета роботи: дослідити алгоритми визначення якості коду.

10.1. Теоретичні відомості

Метрики складності програм прийнято розділяти на три основні групи:

– метрики розміру програм;

– метрики складності потоку керування програм;

– метрики складності потоку даних програм.

Метрики *першої групи* базуються на визначенні кількісних характеристик, пов'язаних з розміром програми, і відрізняються відносною простотою. До найбільш відомих метрик даної групи належать число операторів програми, кількість рядків вихідного тексту, набір метрик Холстеда. Метрики цієї групи орієнтовані на аналіз вихідного тексту програм. Тому вони можуть використовуватися для оцінення складності проміжних продуктів розробки.

Метрики *другої групи* базуються на аналізі керуючого графа програми. Представником цієї групи є метрика МакКейб. Керуючий граф програми, який використовують метрики даної групи, може бути побудований на основі алгоритмів модулів. Тому метрики другої групи можуть застосовуватися для оцінення складності проміжних продуктів розробки.

Метрики *третьої групи* базуються на оціненні використання, конфігурації і розміщення даних у програмі. В першу чергу це стосується глобальних змінних. До даної групи належать метрики Чепіна.

Кількісні метрики (Розмірно-орієнтовані метрики)

LOC-оцінка (Lines Of Code)

Розмірно-орієнтовані метрики прямо вимірюють програмний продукт і процес його розробки. Грунтуються такі метрики на LOC-оцінках. Цей вид метрик побічно вимірює програмний продукт і процес його розробки. Замість

підрахунку LOC-оцінок при цьому розглядається не розмір, а функціональність або корисність продукту.

Найбільшого поширення в практиці створення програмного забезпечення отримали розмірно-орієнтовані метрики. В організаціях, зайнятих розробкою програмної продукції для кожного проекту прийнято реєструвати такі показники:

– загальні трудовитрати (в людино-місяцях, людино-годинах);
– обсяг програми (в тисячах рядків вихідного коду -LOC);
– вартість розробки;
– обсяг документації;
– помилки, виявлені протягом року експлуатації;
– кількість людей, які працювали над виробом;
– термін розробки.

На основі цих даних зазвичай підраховуються прості метрики для оцінення продуктивності праці (KLOC/людино-місяць) і якості виробу.

Ці метрики не універсальні і спірні, особливо це стосується до такого показника як LOC, що істотно залежить від використовуваної мови програмування.

Кількість рядків вихідного коду (Source Lines of Code – SLOC) є найбільш простим і поширеним способом оцінення обсягу робіт за проєктом. Дана метрика була спочатку розроблена для оцінення трудовитрат за проєктом. Однак через те, що одна і та ж функціональність може бути розбита на кілька рядків або записана в один рядок, метрика стала практично непридатною з появою мов, в яких в один рядок може бути записано більше однієї команди. Тому розрізняють логічні і фізичні рядки коду. Логічні рядки коду — це кількість команд програми. Даний варіант опису так само має свої недоліки, оскільки надто залежить від використовуваної мови і стилю програмування

Залежно від того, яким чином враховується подібний код, виділяють два основних показника SLOC:

– Кількість «фізичних» рядків коду – SLOC (використовувані абревіатури LOC, SLOC, KLOC, KSLOC, DSLOC) – визначається як загальне число рядків вихідного коду, включаючи коментарі і порожні рядки (при вимірюванні показника на кількість порожніх рядків, як правило, вводиться обмеження — при підрахунку враховується кількість порожніх рядків, яка не перевищує 25 % загального числа рядків в вимірюваному блоці коду).

– Кількість «логічних» рядків коду – SLOC (використовувані абревіатури LSI, DSI, KDSI, де «SI» – source instructions) – визначається як кількість команд і залежить від використовуваної мови програмування. У тому випадку, якщо язик не допускає розміщення кількох команд на одному рядку, то кількість «логічних» SLOC буде відповідати числу «фізичних», за винятком числа порожніх рядків і рядків коментарів. У тому випадку, якщо мова програмування підтримує розміщення кількох команд на одному рядку, то один фізичний рядок має бути врахований як кілька логічних, якщо він містить більше однієї команди мови.

Для метрики SLOC існує велика кількість похідних, покликаних отримати окремі показники проекту, основними серед яких є:

– число порожніх рядків;

– число рядків, що містять коментарі;

– відсоток коментарів (відношення рядків коду до рядків коментаря, похідна метрика стилістики);

– середнє число рядків для функцій (класів, файлів);

– середнє число рядків, що містять вихідний код для функцій (класів, файлів);

– середнє число рядків для модулів.

Метрика зрозумілості коду

Найбільш простою метрикою стилістики та зрозумілості є оцінення рівня коментування програми F:

$$F = N_{\text{ком}} / N_{\text{стр}},$$

де $N_{ком}$ – кількість коментарів у програмі;

$N_{стр}$ – кількість рядків або операторів початкового тексту.

Таким чином, метрика F відображає насиченість програми коментарями.

Виходячи з практичного досвіду, прийнято вважати, що $F = 0.1$, тобто на кожні десять рядків програми має припадати мінімум один коментар. Як показують дослідження, коментарі розподіляються по тексту програми нерівномірно: на початку програми їх надлишок, а в середині або в кінці — недостатньо. Це пояснюється тим, що на початку програми, як правило, розташовані оператори опису ідентифікаторів, що вимагають більш «щільного» коментування. Крім того, на початку програми також розташовані «шапки», що містять загальні відомості про виконавця, характер, функціональне призначення програми і т.п. Така насиченість компенсує недолік коментарів у тілі програми, і тому наведена формула недостатньо точно відображає коментування функціональної частини тексту програми.

Більш вдалий варіант, коли вся програма розбивається на n рівних сегментів і для кожного з них визначається F_i:

$$F_i = sign(N_{ком}/N_{стр} - 0.1), \text{ при цьому } F = \sum_{i=1}^{n} F_i \ .$$

Рівень коментування програми вважається нормальним, якщо виконується умова: $F = n$. В іншому випадку будь-який фрагмент програми доповнюється коментарями до нормального рівня.

Необхідно підкреслити, що стилістика і зрозумілість програм тісно пов'язана і з розміром, і зі складністю програм. Тому не слід забувати про деяку умовність угруповання метрик програм.

Метрики складності потоку керування програми

Даний клас метрик, що базується на аналізі керуючого графа програми, називається метрикою складності потоку керування програм.

Нехай подається деяка програма. Для даної програми будується орієнтований граф, що містить лише один вхід і один вихід, при цьому вершини графа співвідносять з тими ділянками коду програми, в яких є лише послідовні обчислення і відсутні оператори розгалуження і циклу, а дуги співвідносять з переходами від блоку до блоку і гілками виконання програми. Умова при побудові даного графа: кожна вершина досяжна з початкової, і кінцева вершина досяжна з будь-якої іншої вершини.

Найпоширенішою оцінкою, базованою на аналізі отриманого графа, є цикломатична складність програми (цикломатичне число Мак-Кейба). Вона визначається як $V(G) = e - n + 2p$, де e – кількість дуг, n – кількість вершин, p – число компонент зв'язності. Число компонентів зв'язності графа можна розглядати як кількість дуг, які необхідно додати для перетворення графа в сильно зв'язний. Сильно зв'язним називається граф, будь-які дві вершини якого взаємно досяжні. Для графів коректних програм – графів, які не мають недосяжних від точки входу ділянок і «висячих» точок входу і виходу, сильно зв'язний граф, як правило, виходить шляхом замикання дугою вершини, що позначає кінець програми, на вершину, що позначає точку входу в цю програму. По суті $V(G)$ визначає число лінійно незалежних контурів в сильно зв'язковому графі. Так що в коректно написаних програмах $p = 1$, і тому формула для розрахунку цикломатичної складності набуває вигляду:

$$V\,(G) = e - n + 2.$$

На жаль, дана оцінка не здатна розрізняти циклічні і умовні конструкції. Ще одним істотним недоліком подібного підходу є те, що програми, подані тими самим графами, можуть мати абсолютно різні за складністю предикати (предикат – логічне вираження, що містить хоча б одну змінну).

Існує значна кількість модифікацій показника цикломатичної складності:

– «Модифікована» цикломатична складність – розглядає не кожне розгалуження оператора множинного вибору (switch), а весь оператор як єдине ціле.

– «Сувора» цикломатична складність – включає логічні оператори.

– «Спрощене» обчислення цикломатичної складності – передбачає обчислення не на основі графа, а на основі підрахунку керуючих операторів.

Метрики складності потоку керування даними

Метрика Чепіна: суть методу полягає в оціненні інформаційної міцності окремо взятого програмного модуля за допомогою аналізу характеру використання змінних зі списку введення-виведення.

Вся безліч змінних, складових списку введення-виведення, розбивається на 4 функціональні групи:

1) P – вводяться змінні для розрахунків і для забезпечення виведення;

2) M – модифікуються, або створюються всередині програми змінні;

3) C – змінні, що беруть участь в управлінні роботою програмного модуля (керуючі змінні);

4) T – які не використовуються в програмі («паразитні» змінні).

Оскільки кожна змінна може виконувати одночасно кілька функцій, необхідно враховувати її в кожній відповідній функціональній групі.

Метрика Чепіна:

$$Q = a_1 * P + a_2 * M + a_3 * C + a_4 * T,$$

де a_1, a_2, a_3, a_4 – вагові коефіцієнти.

Вагові коефіцієнти використані для відображення різного впливу на складність програми кожної функціональної групи. На думку автора метрики, найбільшу вагу, що дорівнює 3, має функціональна група C, тому що вона впливає на потік управління програми. Вагові коефіцієнти інших груп розподіляються таким чином: $a_1 = 1, a_2 = 2, a_4 = 0.5$. Ваговий коефіцієнт групи T НЕ

дорівнює 0, оскільки «паразитні» змінні не збільшують складність потоку даних програми, але іноді ускладнюють її розуміння. З урахуванням вагових коефіцієнтів:

$$Q = P + 2M + 3C + 0.5T$$

Обфускація коду

Обфускація – це приведення початкового коду або виконуваного програмного коду до вигляду, який зберігає його функціональність, але ускладнює аналіз, розуміння алгоритму роботи і модифікації при декомпіляції.. Обфускатор – це програма, за допомогою якої виконується обфускація.

«Заплутування» коду може здійснюватися на рівні алгоритму, початкового коду та/або асемблерного коду. Для створення заплутаного асемблерного коду можуть використовуватися спеціалізовані компілятори, які використовують неочевидні або недокументовані можливості середовища виконання програми.

Цілі обфускації:

– Ускладнення декомпіляції/зневадження та вивчення програм з метою виявлення функціональності.

– Ускладнення декомпіляції пропрієтарних програм з метою запобігання зворотної розробки або обходу DRM і систем перевірки ліцензій.

– Порушення авторських прав програмістів і приховування авторства. Парадокс у тому, що використовується це переважно в пропрієтарних програмах.

– Оптимізація програми з метою зменшення розміру працюючого коду і (якщо використовується мова, яка не компілюється) прискорення роботи.

– Демонстрація неочевидних можливостей мови і кваліфікації програміста (якщо проводиться вручну, а не інструментальними засобами).

Недоліки обфускації:

– Втрата гнучкості коду. Код після обфускації може стати більш залежним від платформи або компілятора.

– Складності зневадження. Обфускатор не дає сторонній особі з'ясувати, що робить код, але й не дає розробнику зневаджувати його. При зневадженні доводиться відключати обфускатор.

– Недостатня безпека. Хоча обфускація допомагає зробити розподілену систему безпечнішою, не варто обмежуватися тільки нею. Обфускація - це безпека через приховування. Жоден з існуючих обфускаторів не гарантує складності декомпіляції і не забезпечує безпеки на рівні сучасних криптографічних схем. Цілком імовірно, що ефективний захист неможливий (принаймні в деякому конкретному класі вирішуваних завдань).

– Помилки в обфускаторах. Сучасний обфускатор - складний програмний комплекс. Найчастіше в обфускатори, незважаючи на ретельне проєктування і тестування, проникають помилки. Так що є ненульова ймовірність, що код, який пройшовший через обфускатор взагалі, не буде працювати. І чим складніше програма, що розробляється, тим більше ця ймовірність.

– Виклик класу за іменем. Більшість мов з проміжним кодом можуть створювати або викликати об'єкти по іменах їх класів. Сучасні обфускатори дозволяють зберегти зазначені класи від перейменування, однак подібні обмеження скорочують гнучкість програм.

10.2. Індивідуальне завдання

Використовуючи код клієнта з лабораторної роботи «Геш-Дерева», визначити такі показники якості коду:

– рівень якості програмування;
– складність розуміння програми;
– трудомісткість кодування програми;
– цикломатичне число Мак-Кейба;
– метрика Чепіна.

Метрики потрібно виконувати для:

– source code;

– декомпільованого скопільованого коду;

– декомпільованого скопільованого обфускованого (наприклад, stinger) коду.

Зробити висновки впливу компіляції та обфускації на якість коду.

При виконанні роботи бажано використовувати:

– Roslyn (https://github.com/dotnet/roslyn) (тільки для .net).

– Antrl (https://www.antlr.org).

Будь-які інші бібліотеки для аналізу синтаксичного дерева потрібно обговорювати з викладачем.

Контрольні запитання

1. На які групи поділяються метрики?
2. В чому суть LOC-оцінення?
3. За якими показниками можливо провести SLOC-оцінення?
4. Як обчислюється метрика зрозумілості коду?
5. При яких умовах метрика зрозумілості коду дає достатньо точний результат?
6. Що таке цикломатична складність програми?
7. Як отримати сильно зв’язний граф для графів коректних програм?
8. Які недоліки оцінення за показником цикломатичної складності ви знаєте?
9. Які існують модифікації показника цикломатичної складності?
10. У чому полягає суть метрики Чепіна?
11. На які функціональні групи можна поділити змінні, що входять до складу списку введення-виведення?
12. Що таке обфускація?
13. Як можна здійснити обфускацію?
14. Для чого робиться обфускація програмного коду?
15. Назвіть недоліки використання обфускації для захисту інформації.

СПИСОК ЛІТЕРАТУРИ

1. Страуструп, Бьерн. Язык программирования С++. Специальное издание, /Б. Страуструп; пер. с англ.; под ред. Н. Н. Мартынова.: Т. 1: Основные алгоритмы. – Москва : Бином, 2011. – 1035 с.

2. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Safford "Documenting Software Architectures. Views and Beyond" Addison-Wesley Publishing 2008.

3. Len Bass, Paul Clements, Rick Kazman, "Software Architecture in Practice", Addison-Wesley Professional, Second Edition, 2003.

4. Eric J. Braud "Software Engineering: An Object-Oriented Perspective", Wiley Computer Publishing, 2001.

5. Richard N. Tailor, Nenad Medvidovic, Eric M. Dashfy, "Software Architecture. Foundations, Theory, and Practice", Wiley, 2010.

6. Jonh Viega, Gary McGraw "Building secure software. How to Avoid Security Problems the Right Way", Addison-Wesley Publishing Company, 2005

7. Gary McGraw "Software Security. Building Security in", Addison-Wesley Publishing Company, 2006

8. Greg Hoglund, Gary McGraw "Exploiting Software. How to Break Code", Addison-Wesley Publishing Company, 2004

ДЛЯ ПРИМІТОК

Навчальне видання

ДАВИДОВ Вячеслав Вадимович
СЕМЕНОВ Сергій Геннадійович
КУЧУК Ніна Георгіївна
БУЛЬБА Сергій Сергійович

СУЧАСНІ ТЕХНОЛОГІЇ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ
Навчально-методичний посібник
для студентів спеціальності 123
«Комп'ютерна інженерія»

Відповідальний за випуск проф.
Роботу до видання рекомендував проф. Микола Заполовський
Редактор О.І Шпільова

План 2021 р., поз. 25

Видавничий центр НТУ «ХПІ»
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Фрунзе, 21

Підписано до друку 01.06.2021. Формат 60х84 1/16. Папір офсетний.
Друк – ризографія. Гарнітура Times New Roman. Ум. друк. арк. 6,51.
Наклад 100 прим. Зам. № 06-21. Ціна договірна.

Надруковано ТОВ «Видавництво «Форт»»
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, витівників
і розповсюджувачів видавничої продукції,
серія ДК №7107 від 20.07.2020р.
61023, м. Харків, а/с 10325. Тел. +38 (057) 714-09-08