МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРИЗОВАНИХ СИСТЕМ ЗАХИСТУ ІНФОРМАЦІЇ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____ С.В. Казмірчук

«_____»_____ 20__ р.

На правах рукопису
УДК 004.056.5:510.22(043.3)

МАГІСТЕРСЬКА АТЕСТАЦІЙНА РОБОТА

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
«МАГІСТР»

Тема: Програмний модуль системи захисту ядра Linux

Автор: Я.В. Яценко

Науковий керівник: к.т.н., доц. С.Є. Карловський

Нормоконтролер: асист.

Київ 2020

ВСТУП

В наш час все більшого розповсюдження набирає використання мобільних телефонів. З часом вони стають все складнішими і досконалішими. Сучасні телефони можуть без особливих проблем конкурувати із комп'ютерами, вони мають змогу запускати повноцінні операційні системи. Разом із уподібненням сучасних смартфонів персональних комп'ютерів вони отримують практично той самий набір загроз і можливостей компрометації безпеки, що й звичайні комп'ютери і операційні системи.

Ми живемо у все більш мобільному світі. Відповідно до останніх спостережень[16] 75% компаній дозволяють працівникам використовувати персональні пристрої для роботи. Отже, працівники можуть зберігати конфіденційні дані на них, та їх втрата або оприлюднення може нашкодити як і самим користувачам так і компанії.

Користувач може загрузити застосунок із недовіреного джерела і зовсім не підозрювати, що цей застосунок може бути зловмисним троянським застосунком, який використовує вразливості операційної системи смартфону для, наприклад підвищення привілеїв, а потім краде, змінює конфіденційні дані.

Серед розпювсюджених загроз, які використовуюься на практиці зловмисниками - злам смартфона, використовуючи віддалений зв'язок, стеження за діяльністю власника смартфона, а також руткіти й звичайні віруси.

Для знаходження вірусів та руткітів на стаціонарних комп’ютерах зазвичай використовують антивірусні програми або інтроспекцію віртуальних машин, але сучасні віруси можуть дізнатися про існування в системі аналізуючої програми й зупинити свою діяльність, а технологія віртуалізації не настільки розвинута в сфері мобільних пристроїв, щоб її можна було ефективно використовувати. Більш того, самі антивірусні програми можуть мати вразливості, які може використовувати вірусна програма, оскільки вони працюють на одному рівні.

Технологія віртуалізації [12] може покращити прозорість систем віртуалізації і її ефективність на мобільний пристроях; однак цей підхід все ще залишає артефакти, які можуть бути легко виявлені зловмисним програмним забезпеченням [4, 5, 6].

Тому в даній роботі пропонується інший метод знаходження шкідливого програмного забезпечення в системі на мобільному пристрої, використовуючи платформо-специфічні особливості ARM пристроїв.

Актуальність роботи полягає в тому, що сфера апаратних пристроїв із операційною ситемою Linux швидко розвивається, а разом з цим і розробники зловмисного програмного забезпечення вдосконалюють методи атак. Дана робота пропонує метод захисту інформації від руткітв, який використовує – довірене середовище виконання (Trusted Execution Environment, trustzone), а також не залишає артефактів, які дають змогу зловмисному програмному забезпеченню виявити віртуальне середовище.

Метою роботи покращення безпеки для апаратних засобів із операційною системою на основі ядра Linux, це досволить блокувати роботу руткітів або не дозволити їм бути запущеними.

Завданням роботи є:

1) Огляд та аналіз існуючих вразливостей у сфері інформаційної безпеки для апаратних засобів із операційною системою на основі ядра Linux.
2) Аналіз існуючих методів захисту в Linux подібній операційній системі.
3) Розроблення методу динамічного аналізу завантажуваних модулів.
4) Розробка тестового застосунку, що втілює п.4.

Об'єктом дослідження системи та процеси захисту інформації.

Предметом дослідження апаратні засоби із операціною системою Linux.

Новизна роботи зумовлюється тим, що в роботі запропоновано метод

визначення зловмисного програмного забезпечення в операційній системі мобільного пристрою, який відрізняється використанням Trusted Execution Environment.

Практичне значення полягає в тому, що результати роботи можуть бути використані на практиці в реальній операційній системі для вдосконалення її безпеки. Дане рішення можна використовувати в якості динамічного аналізатора зловмисного програмного забезпечення.

1 ОГЛЯД ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX, ОСОБЛИВОСТІ ARM АРХІТЕКТУРИ, ОСНОВНІ

1.1 Linux-подібна операційна система

Операційна система Linux - це набір програм, який керує комп'ютером, здійснює зв'язок між вами і комп'ютером та забезпечує вас інструментальними засобами, щоб допомогти вам виконати вашу роботу. Розроблена, щоб забезпечити легкість, ефективність і гнучкість програмного забезпечення, система Linux[1] має кілька корисних функцій:

- оновна мета системи - це виконувати широкий спектр завдань і програм;
- інтерактивне оточення, яке дозволяє вам зв'язуватися безпосередньо з комп'ютером і отримувати негайно відповіді на ваші запити і повідомлення;
- багатокористувацької оточення, яке дозволяє вам поділяти ресурси комп'ютера з іншими користувачами без зменшення продуктивності. Цей метод називається поділом часу. Система Linux взаємодіє з користувачами по черзі, але так швидко, що здається, що взаємодіє з усіма користувачами одночасно;
- багатозадачне оточення, яке дозволяє вам виконувати більш одного завдання в одне і теж час.

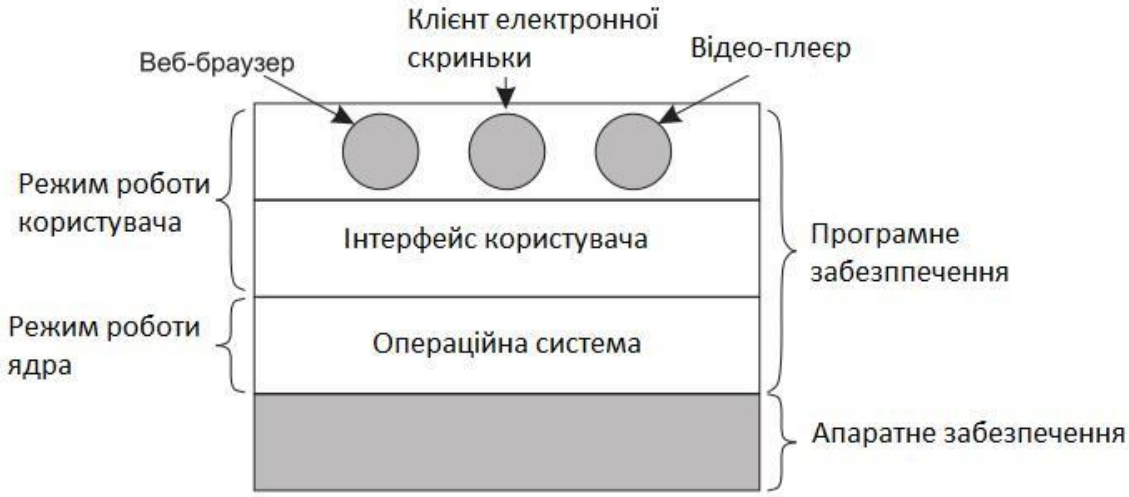

Рис. 1.1 - Операційна системи в структурі програмного забезпечення

Система Linux має 4 основних компоненти:

- kernel - це програма, яка утворює ядро операційної системи; вона координує внутрішні функції комп'ютера (такі як розміщення системних ресурсів);
- shell - це програма, яка здійснює зв'язок між вами і ядром, інтерпретуючи і виконуючи ваші команди. Так як вона читає ваш введення і посилає вам повідомлення, то описується як інтерактивна;
- commands - це імена програм, які комп'ютер повинен виконати. Пакети програм називаються інструментальними засобами. Система Linux забезпечує інструментальними засобами для таких завдань як створення і зміна тексту, написання програм, розвиток інструментарію програмного забезпечення, обмін інформацією з іншими за допомогою комп'ютера;
- file system - це набір усіх файлів, можливих для вашого комп'ютера. Вона допомагає вам легко зберігати і відшукувати інформацію.

Ядро контролює доступ до комп'ютера, управляє пам'яттю комп'ютера, обслуговує файлову систему і розподіляє ресурси комп'ютера серед користувачів.

shell - це програма, яка дозволяє вам зв'язуватися з операційною системою. shell зчитує команди, які ви вводите, і інтерпретує їх як запити на виконання інших програм, на доступ до файлу або забезпечення виведення. shell також є потужним мовою програмування, не схожим на мову программіровані Сі, який опеспечівает умовне виконання і управління потоками даних.

Програма - це набір інструкцій для комп'ютера. Програми, які можуть бути виконані комп'ютером без попередньої трансляції, називаються виконуваними програмами або командами. Як звичайному користувачеві системи Linux вам доступна безліч стандартних програм та інструментальних засобів. Якщо ви використовуєте систему Linux для написання програм і розвитку програмного

забезпечення, то ви також можете задіяти системні виклики, підпрограми та інші інструментальні засоби. І, звичайно, будь-яка написана вами програма буде у вашому розпорядженні.

Зовнішнє коло системи Linux утворюють програми і інструментальні засоби системи, розділені на категорії функціонально. Ці функції включають:

- програмне оточення - кілька програм системи Linux, що встановлюють дружнє програмне оточення, що забезпечує інтерфейси між системою і мовами програмування і використання обслуговуючих програм;
- обробка текстів - система забезпечує програми, такі як строковий і екранний редактори, для створення і зміни текстів, орфографічну програму перевірки для виявлення помилок орфографії, і необов'язковий форматер тексту для створення високоякісних копій, які підходять для публікацій;
- організація інформації - система надає багато програм, які дозволяють вам створювати, організовувати і видаляти файли і каталоги;
- обслуговуючі програми - інструментальні засоби, що створюють графіку і виконують обчислення;
- електронний зв'язок - кілька програм (наприклад, mail) надають вам можливість передавати інформацію іншим користувачам і в інші системи Linux.

Щоб ваш запит був зрозумілий системі Linux ви повинні ввести кожну команду в коректному форматі або синтаксисі командного рядка. Цей синтаксис визначає порядок, в якому ви вводите компоненти командного рядка. І ви повинні розташувати всі складові частини командного рядка в необхідному синтаксисом порядку, інакше shell не зможе інтерпретувати ваш запит.

Приклад синтаксису командного рядка:

command option (s) argument (s) <CR>

Для кожної командного рядка системи Linux ви повинні ввести як мінімум два компоненти: ім'я команди і клавішу <RETURN>[13]. Командний рядок може також містити ключі і аргументи. У зазначеному прикладі синтаксису командного рядка:

- command - це ім'я програми, яку ви хочете виконати;
- option - ключі, які вказують як запустити команду;
- argument - вказує на дані, які ця команда обробляє, зазвичай це ім'я каталогу або файлу.

Файлова система є наріжним каменем операційної системи Linux. Вона забезпечує логічний метод організації, відновлення та управління інформацією. Файлова система має ієрархічну структуру.

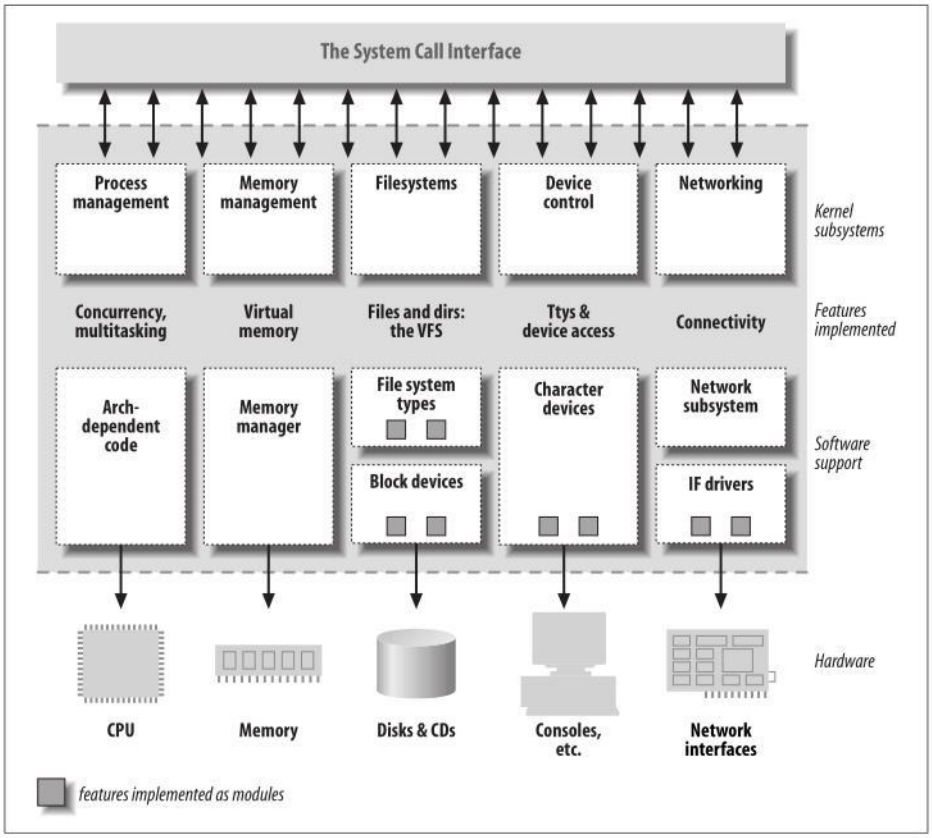

Рис. 1.2 – Частини ядра Linux.

Файл, який є основною одиницею системи Linux, може бути: звичайним файлом, довідником, спеціальним файлом або символічним каналом зв'язку.

Звичайні файли є набором символів. Звичайні файли використовуються для зберігання будь-якої інформації. Вони можуть містити тексти для листів або звітів, коди програм, які ви написали, або команди для запуску ваших програм. Одного разу створивши звичайний файл ви можете додати потрібний матеріал в нього, видалити матеріал з нього, або видалити файл цілком.

Довідники є супер-файлами, які можуть містити файли або інші довідники. Зазвичай файли, що містяться в них, встановлюють відносини будь-яким способом. Наприклад, довідник, названий sales може зберігати файли, що містять цифри щомісячних продажів, названі jan, feb, mar, і т.д. Ви можете створити каталоги, додати або видалити файли з них або видалити каталоги.

Всі довідники, які ви створюєте, будуть розміщені у вашому власному довіднику. Цей довідник призначається вам системою під час входу в систему. Ніхто крім привілейованих користувачів не може читати або записувати файли в цей довідник без вашого дозволу і ви визначаєте структуру цього довідника.

Система Linux також містить кілька довідників для власного використання. Структура цих довідників аналогічна у всіх системах Linux. Цей довідник, що включає в себе кілька системних довідників, розміщується безпосередньо під довідником root. Довідник root (позначений /) є вихідним в файлової структурі Linux. Всі довідники і файли ієрархічно розташовуються нижче.

Спеціальні файли відповідають фізичних пристроїв, таким як термінал, накопичувач на комп'ютері, магнітна стрічка або канал зв'язку. Ситема читає і записує з / в спеціальні файли також як і в звичайні файли. Однак запити системи на читання і запис не призводять в дію нормальний механізм доступу до файлу.

Замість цього вони активізують драйвер пристрою, пов'язаний з файлом, приводячи, можливо, в дію головки диска або магнітної стрічки.

Деякі операційні системи вимагають від вас визначити тип файлу і використовувати його певним способом. Від цього буде залежати те, як будуть файли зберігатися, тому що файли можуть бути послідовними, двійковими або довільної вибірки

Для системи Linux всі файли однакові. Це робить файлову структуру Linux легкою у використанні. Наприклад, вам немає необхідності вказувати вимоги до пам'яті для ваших файлів, тому що система автоматично це зробить для вас. Або якщо вам або написаної вами програмі необхідний доступ до певного пристрою (наприклад, принтера) ви вказуєте пристрій також як будь-який з ваших файлів. В системі Linux існує тільки один інтерейса для всього вашого введення і виведення для вас; це спрощує ваше взаємодія з системою.

1.2 Завантажувані модулі ядра

Системне ядро Linux здатне до модифікації за рахунок розширення функціональних можливостей. Це досягається декількома способами, але найоптимальнішим є підключення завантажувальних модулів ядра. Цей спосіб дозволяє вести розробку потрібного функціоналу для ядра незалежно від самого ядра. Процес підключення модулів до ядра, як правило, ніяких труднощів не викликає (на відміну від конфігурації і компіляції). Якщо звичайно модулі розроблені грамотно і належним чином протестовані. Системним адміністраторам варто пам'ятати, що якщо потрібно «розширити» ядро під необхідний функціонал, то в першу чергу слід скористатися варіантом у вигляді завантажуваних модулів ядра.

Модуль ядра[2] - це певний код, який може бути завантажений або вивантажений ядром в міру необхідності. Модулі розширюють функціональні можливості ядра без необхідності перезавантаження системи. Наприклад, одна з різновидів модулів ядра, драйвери пристроїв, дозволяють ядру взаємодіяти з апаратурою комп'ютера. При відсутності підтримки модулів нам довелося б писати монолітні ядра і додавати нові можливості прямо в ядро. При цьому, після додавання в ядро нових можливостей, довелося б перезавантажувати систему.

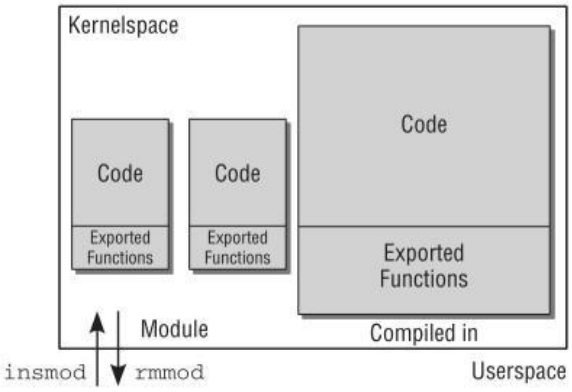

Рис. 1.3 – Ядро Linux із завантаженим модулем.

За допомогою завантажувальних модулів ядра досить просто додавати в систему драйвери, підтримку нових технологій і можливостей для архітектури Linux. Це не вимагає зачіпання компцонентів, вже включених в ядро. Не потрібно модифікувати вихідний код ядра і возитися з його компіляцією. До того ж, це економить пам'ять і дисковий простір під саме ядро. Адже можна завантажувати тільки ті модулі, які використовуються фактично.

Однак серед всіх незаперечних і вагомих плюсів у використанні завантажувальних модулів ядра, існують і певні мінуси - необхідна стовідсоткова впевненість в тому, що конкретний завантаження модуль надійний і стабільний. Інакше при завантаженні і вивантаженні цього модуля велика ймовірність порушити працездатність ядра. І його потрібно буде конфігурувати заново.

Серед всіх UNIX-подібних операційних систем найбільш пристосованими для роботи з завантажуються модулями ядра (а також підтримують найширший спектр обладнання та апаратних платформ, в тому числі і новинок) є системи Linux і Solaris.

Як вже було зазначено, Linux здатна надати системним адміністраторам досить широкі можливості для роботи з завантажуються модулями ядра. Так наприклад в Linux будь-який системний компонент можливо зробити завантажуваних модулем за винятком двох системних об'єктів: драйвера пристрою, що використовується для роботи кореневої файлової системи (FS) і драйвера миші типу PS / 2 [14].

Всі завантажувані модулі зберігаються (і встановлюються) в каталог / lib / modules / версія_ядра. Тут версія_ядра може визначена командою:

$ uname -r

Для то, щоб отримати список завантажених і використовуваних в даний момент модулів ядра, можна скористатися командою lsmod:

$ sudo lsmod
Module Size Used by
ipmi_devintf 13064 2
ipmi_si 36648 1
ipmi_msghandler 31848 2 ipmi_devintf,ipmi_si
iptable_filter 6721 0
ip_tables 21441 1 iptable_filter

Як видно з даного виводу, в системі налаштована підтримка брандмауера iptables, а також використовуються модулі для інтелектуального управління платформою IMPI - Intelligent Platform Management Interface. У колонці «Size» вказується розмір в байтах, а в колонці «Used by» (ким використовується) наводиться кількість зв'язків на даний модуль від інших компонентів, а також список самих цих компонентів.

Навіть якщо системне ядро було налаштоване і скомпільовано з настройками і комплектністю за замовчуванням, то при використанні команди lsmod, як правило буде виведений досить довгий список модулів. Якщо відома точна або приблизна ім'я модуля, який необхідно перевірити (використовується він чи ні), то разом з командою lsmod зручно використовувати і команду grep:

$ sudo lsmod | grep amdgpu
amdgpu 3166208 92
amdchash 16384 1 amdgpu
amd_shed 24576 1 amdgpu
i2c_algo_bit 16384 1 amdgpu
amdttm 110592 1 amdgpu
amdkcl 28672 4 amdkfd,amd_shed,amdttm,amdgpu
drm_kms_helper 172032 1 amdgpu
drm 401408 24 drm_kms_helper,amdkfd,amd_shed,amdttm,amdgpu,amdkcl

В даному виводі показані результати для використовуваних ядром модуляхдрайвера amdgpu.

1.3 Завантаження та підключення модулів ядра Linux

Суть завантаження модулю ядра полягає у виконнані функції load_module ядра операційної системи:

```
/* Allocate and load the module: note that size of section 0 is always
   zero, and we rely on this for optional sections. */
static int load_module(struct load_info *info, const char __user *uargs,
                       int flags)
{
        struct module *mod;
        long err;
        char *after_dashes;

        err = module_sig_check(info, flags);
```

Рис. 1.4 – Визначення load_module

При виконанні функції load_module виконуються перевірки підпису модулю, якщо така конфігурація присутня та перевірка заголовку ELF файлу.

Нехай є завантажуваний модуль ядра для деякого пристрою. Наприклад є файл модуля somedevice.ko для пристрою somedevice. Щоб завантажити цей модуль для ядра прямо під час роботи системи слід використовувати команду insmod:

$ sudo insmod /путь/к/файлу/somrdevice.ko

За допомогою цієї ж команди можна також і задавати конфігураційні параметри, необхідні для необхідних або специфічних режимів роботи модулів:

$ sudo insmod /путь/куда/somedevice.ko io=0xXXX irq=X

Для видалення модуля потрібно або перезапустити систему або вказати в явному запиті вивантажуваний модуль за допомогою команди rmmod:

$ rmmod somedevice

Слід зауважити, що не дивлячись на те, що команду rmmod хоч і можна використовувати в будь-який момент часу, однак модуль може бути видалений лише в разі, якщо з ним не пов'язано жодної діючої посилання, які вказуються в стовпці «Used by» виведення команди lsmod .

Щоб кожен раз після перезапуску системи не завантажувати необхідні модулі, в Linux існує можливість підключати їх автоматично за допомогою файлу /etc/modprobe.conf і відповідної команди modprobe[2]. Ця команда є «обгорткою» команди insmod. Вона здатна визначати порядок завантаження і вивантаження модулів, їх параметри, а також залежно від інших модулів і / або параметрів. Всю цю інформацію команда modprobe читає з файлу /etc/modprobe.conf, який, до речі здатна і сама генерувати.

Щоб згенерувати файл modprobe.conf відповідно до поточного набору завантажених модулів (які необхідно завантажити заздалегідь), потрібно використовувати командух:

$ sudo modprobe -c

В результаті буде створений досить довгий файл, вміст якого може виглядати приблизно так:

#This file was generated by: modprobe -c

path[pcmcia]=/lib/modules/preferred
path[pcmcia]=/lib/modules/default

path[pcmcia]=/lib/modules/2.6.6
path[misc]=/lib/modules/2.6.6

 # Aliases
alias block-major-1 rd
alias block-major-2 floppy
alias char-major-4 serial
alias char-major-5 serial
alias char-major-6 lp
alias dos msdos
alias plip0 plip
alias ppp0 ppp
options ne io=x0340 irq=9

Ключові слова path визначають розташування конкретних модулів в файлової системі, а aliases задають прив'язку файлових систем, мережевих протоколів, старших номерів блокових і символьних пристроїв до необхідного модулю.

Значення для рядків options адміністратори повинні задавати самостійно, оскільки ці інструкції командою modprobe автоматично не генеруються. Наприклад, щоб задати для модуля пристрою somedevice адресу його введення-виведення, а також вектор переривань, можна це зробити таким чином:

options somedevice io=0xXXX irq=X

Варто відзначити також, що для команди modprobe доступні ключові слова install і remove, за допомогою яких можна вказувати команди, які будуть виконуватися під час завантаження та встановлення модулів ядра відповідно.

Як можна бачити, в процедурі управління модулями немає нічого надто складного, проте це той випадок, коли системним адміністраторам слід проявляти особливу акуратність і обережність, перевіряючи кожен модуль, який планується використовувати в системі.

1.4 Керування пам’яттю процесів у Linux

Базовою одиницею в організації пам'яті для систем UNIX / Linux є сторінка пам'яті. Що володіє розміром від 4 Кбайт, якому відповідав би обсяг фізичного простору в оперативній або віртуальної (область підкачки на диску або іншому пристрої для збереження) пам'яті. При запуску процесів, вони запитують у системи (т. е. у ядра за допомогою відповідних системних викликів) пам'ять для своєї роботи. А у відповідь на це ядро виділяє для них достатню кількість сторінок пам'яті. Віртуальна пам'ять або як її ще називають, «резервний ЗП» для сторінок пам'яті. Які містять, наприклад, вихідний текст виконуваного додатка, є звичайними виконуваними файли на диску. Так само як і для інших файлів даних резервним ЗП є самі файли. Інформація про те як взаємопов'язані сторінки фізичної і віртуальної пам'яті зберігається у відповідних таблицях сторінок пам'яті.

Рис. 1.5 – Зв'язок між віртуальними адресами і адресами фізичної пам'яті.

Для роботи з пам'яттю в Linux (як і в інших UNIX-подібних системах) характерно таке явище як «сторінковий обмін» (paging)[3]. Воно полягає в тому, що ядро виділяє процесам стільки пам'яті, скільки їм необхідно. В тому сенсі, щоб її (пам'яті) завжди вистачало. Це досягається за рахунок розширення фізичної пам'яті за рахунок віртуальної, тобто «підкачки». Оскільки виконання процесів має відбуватися в реальній фізичній пам'яті. Ядро постійно переміщує сторінки пам'яті процесів між фізичною та віртуальною пам'яттю. Забігаючи наперед, слід зазначити, що у віртуальній пам'яті зберігаються «неактивні» сторінки. Які не задіяні процесом в даний момент, але необхідні йому для повноцінної роботи згодом.

Перше, на що слід звернути увагу, це те, що ядро намагається управляти пам'яттю таким чином, щоб недавно використовувані процесом сторінки знаходилися в фізичній пам'яті. І в свою чергу, «неактивні» або рідко використовувані сторінки переміщуються і зберігаються в віртуальної пам'яті в області «підкачки». Такий механізм розподілу пам'яті називається LRU (least recently used) - заміщення найбільш рідко використовуваних сторінок.

Другим найважливішим аспектом в роботі пам'яті є використання кеш-буфера сторінок. Це випливає з роботи алгоритму LRU, який досить складний у своїй

реалізації. Оскільки стежити за всіма зверненнями до сторінок - це в деяких випадках, досить відчутні втрати в продуктивності системи. Використання ж сторінкового кеш-буфера куди простіше в своїй реалізації при тих же самих результатах. До того ж даний підхід має величезний модернізаційний потенціал (на відміну від LRU) і алгоритми аналізу вмісту кеш-буфера (для визначення, які сторінки повинні бути переміщені з віртуальної пам'яті) постійно удосконалюються. Що помітно позначається на продуктивності і ефективності управління пам'яттю.

Коли процесу не вистачає пам'яті, то ядро починає шукати «зайняті» сторінки. Які можна використовувати для «голодуючого» процесу. Зазвичай такими сторінками є ті, що давно не були використовувані. Ядро перевіряє їх на предмет модифікації будь-яким процесом. Для цього існують певні ознаки, при останньому зверненні і якщо зміни були, то такі сторінки позначаються ядром як «брудні». Тобто такі, які ще потрібні процесам. Для повторного використання пам'яті такі сторінки спочатку обов'язково переносяться в віртуальну пам'ять. Всі ж інші сторінки є «чистими». І тому ядро їх використовує для надання іншим або «голодуючим» процесам.

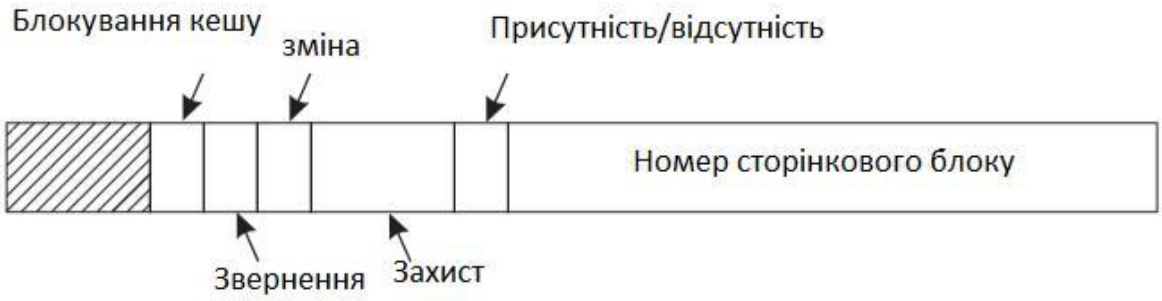

Рис. 1.6 – Запис у таблицю сторінок.

Коли відбувається звернення до сторінок пам'яті, які деякий або довгий час не використовувались, тобто до «неактивних» сторінок. Ядро виконує з ними кілька важливих завдань:

- повертає посилання на ці сторінки у відповідній таблиці сторінок;
- скидає в нульове значення час «невикористання» цих сторінок;
- позначає ці сторінки як «активні».

Зі сторінками, що знаходяться у віртуальній пам'яті не все так однозначно. Справа в тому, що для того, щоб «активізувати» такі сторінки, вони повинні бути попередньо прочитані з диска.

Системне ядро комплектується спеціалізованими модулями. Які містять алгоритми і навіть цілі технології. За допомогою яких система досить ефективно «пророкує», скільки може знадобитися пам'яті при різного ступеня активності і завантаженості процесів. Ці алгоритми мають на меті забезпечення процесів вільною пам'яттю з максимальною ефективністю. Тобто так, щоб процесам якомога рідше доводилося простоювати в «очікуванні» вивантаження чергової сторінки в вільну пам'ять. Таким чином, спостерігаючи за станом сторінкового обміну під час робочого навантаження системи, можна робити висновки про те, чи потрібна їй додаткова пам'ять. Якщо сторінковий обмін інтенсивний - то однозначно слід встановити додаткові модулі ОЗП.

Якщо ж відбувається так, що процесам не вистачає ні реальної фізичної, ні віртуальної пам'яті. Тобто коли пам'ять повністю вичерпана, то система починає завершувати (а точніше знищувати) цілі процеси. Або забороняє створення нових. Звичайно в цьому випадку в першу чергу знищуються найбільш «безболісні» для системи процеси. Однак в таких випадках навіть «на око» і за власними відчуттями видно що вона більшу частину часу витрачає на управління пам'яттю, а не на виконання робочих завдань.

В Linux можна налаштувати параметр, який задає, наскільки швидко ядро має «відбирати» сторінки пам'яті у процесів. Яким вони менш потрібні для процесів, яким вони на даний момент необхідні. Цей параметр міститься в файлі /proc/sys/vm/swappiness і за замовчуванням дорівнює 60. Якщо задати його меншим

значенням (наприклад 0). Те ядро буде забирати сторінки процесу в найостаннішу чергу. Використовуючи замість цього будь-які інші варіанти. Якщо це значення в межах між 60 і 100 сторінки будуть відбиратися у процесів з більш високою ймовірністю. Варіант зі зміною даного параметра насправді говорить про те, що необхідно або знизити навантаження на систему. Адаптувавши її для інших менш продуктивних завдань, або збільшити обсяг ОЗП.

Вся віртуальна пам'ять процесу в Linux поділена на кілька областей (сегментів), різного призначення. Виділяються наступні області[4]:

- сегмент коду, що містить інструкції процесора, який відображається на відповідні області виконуваного файлу. Традиційно позначається як text.
- сегмент ініціалізованих глобальних змінних, які спочатку завантажуються з виконуваного файлу. Позначаються як data.
- сегмент неініціалізованих глобальних змінних, який спочатку заповнений нулями. Традиційно позначається BSS (Block Started by Symbol). Оскільки зберігати в виконуваному файлі нулі не має сенсу, то в ньому зберігається ім'я змінної (symbol) і її розмір, а виділення пам'яті і ініціалізація нулями, відбувається в момент запуску програми.
- сегмент стека - Stack.
- область купи, для динамічного виділення пам'яті функціями malloc (), new ().
- додаткові сегменти коду, даних і BSS для кожної розділяється між процесами бібліотеки, такий як бібліотека libc. Динамічні бібліотеки в Linux мають суфікс в імені файлу - .so, що означає shared object.
- область для відображення в пам'ять файлів.
- область для відображення розділяється пам'яті (shared memory).
- в сучасному Linux в окремі області пам'яті vsyscall і vdso (virtual dynamic shared object) винесено часто викликається код ядра, що не вимагає системних повноважень. Як приклад такого зазвичай призводять виклик gettimeofday().

Винесення коду в віртуальну пам'ять простору допомагає заощадити час, що витрачається на зміну таблиці сторінок під час системного виклику.

Кожна область пам'яті описується структурою, яка зберігає початок і кінець області в віртуальному просторі процесу, позицію в файлі, якщо область є відображенням файлу в пам'ять, права доступу і прапорці, що описують властивості області.

1.5 Стек виконання

Стек виконання - в теорії обчислювальних систем, LIFO-стек, який зберігає інформацію для повернення управління з підпрограм (процедур, функцій) в програму (або підпрограму, при вкладених або рекурсивних викликах) і / або для повернення в програму з обробника переривання (в тому числі при перемиканні задач в багатозадачному середовищі).

```
1  void call(int a)
2  {
3      (void)a;
4  }
5
6  int main(int argc, char **arv)
7  {
8      int a = 5;
9      call(a);
10     return 0;
11 }
```

```
1  call:
2    str fp, [sp, #-4]!
3    add fp, sp, #0
4    sub sp, sp, #12
5    str r0, [fp, #-8]
6    add sp, fp, #0
7    ldmfd sp!, {fp}
8    bx lr
9  main:
10   stmfd sp!, {fp, lr}
11   add fp, sp, #4
12   sub sp, sp, #16
13   str r0, [fp, #-16]
14   str r1, [fp, #-20]
15   mov r3, #5
16   str r3, [fp, #-8]
17   ldr r0, [fp, #-8]
18   bl call
19   mov r3, #0
20   mov r0, r3
21   sub sp, fp, #4
22   ldmfd sp!, {fp, pc}
```

Рис. 1.7 – Приклад стеку виконання.

При виклику підпрограми або виникненні переривання, в стек заноситься адреса повернення - адреса в пам'яті наступної інструкції призупиненої програми і управління передається підпрограмі або підпрограми-обробника. При подальшому вкладеному або рекурсивному виклику, перериванні підпрограми або обробника переривання, в стек заноситься чергова адреса повернення і т. д.

При поверненні з підпрограми або обробника переривання, адреса повернення знімається зі стека і управління передається на наступну інструкцію призупиненої підпрограми[5].

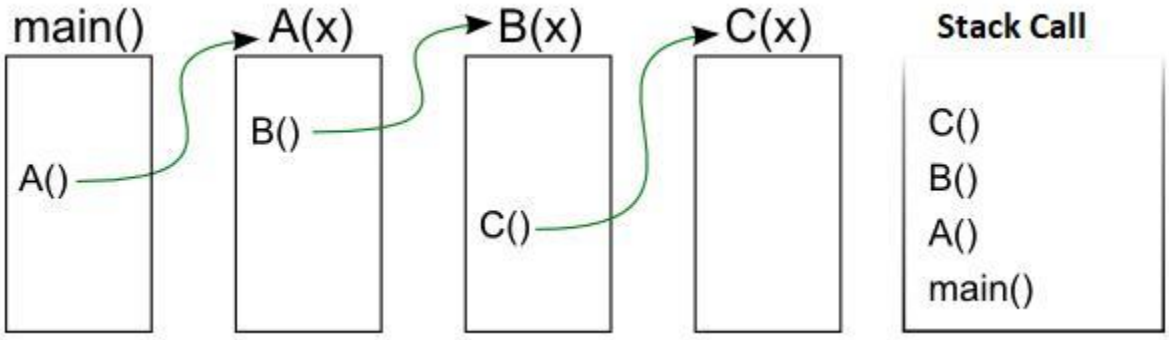

Рис. 1.8 – Приклад стеку виконання.

Стек викликів зазвичай реалізований одним із способів:

- в більшій частині платформ стек розташовується в оперативній пам'яті (або регістровому файлі, як в мікроконтролері Intel 8051), спеціалізований регістр вказує на його вершину;
- замість стека викликів адреса повернення зберігається в спеціалізований регістр, який зберігає один або кілька значень адрес.

При відсутності стека або обмеженості його глибини, вкладені виклики виключені або їх кількість обмежена. При необхідності більшої вкладеності, стек викликів або його розширення можуть бути реалізовані програмно.

Виклик підпрограми і повернення з підпрограм і обробників переривань, як правило виконуються спеціалізованими інструкціями процесора. Крім інструкцій викликів і повернень, процесори часто мають інструкції для використання стека викликів також і під збереження даних - їх переміщення в стек, зняття з стека, модифікації вмісту стека.

Іноді процедури повернення з підпрограми і обробника переривань відрізняються один від одного, і також вимагають різних команд (наприклад, при поверненні з переривання часто необхідно відновити з стека регістр прапорів і / або дозволити обробку конкурентних переривань, яка може автоматично заборонятися при виклику обробника).

При відсутності спеціалізованих інструкцій (в процесорах з RISC-) виклики, повернення та інша робота зі стеком викликів реалізуються звичайними інструкціями роботи з пам'яттю / регістрами і передачі управління.

Стек викликів може використовуватися для різних потреб, але основне його призначення - відстежувати місце, куди кожна з викликаних процедур повинна повернути управління після свого завершення. Для цього при виклику процедури (командами виклику) в стек заноситься адреса команди, наступної за командою виклику («адресаповернення»). По завершенні викликана процедура повинна виконати команду повернення для переходу за адресою з стека.

Крім адрес повернення в стеку можуть зберігатися інші дані, наприклад:

- значення регістрів з їх подальшим відновленням;
- дані стекового кадру мов високого рівня
- аргументи, передані в функцію
- локальні змінні - тимчасові дані функції
- інші довільні дані

1.7 Рівні виконання процесів

Для даної роботи важливо рзглянути рівні виконня, на яких може знаходитись виконавчий процес у системи для процесора архітектури ARM, що імплементує так зване довірене середовище виконання, тобто Trusted Execution Environment.

Тож існує декільки рівнів, на яких може перебувати процес знаходячись на виконанні, до них належать наступні:

- EL0 - найнижчий рівень, виконнання у безпечному світі.
- EL1 - рівень, на якому запускається операційна система працюючої машини.
- EL2 - рівень наглядача. Застосовується для роботи гіпервізору системи. Цей рівень завжди знаходиться у незахищеному стані.
- EL3 - рівень для secure monitor. Він призначений для того, щоб була можливою комунікація і переходи між безпечним та небезпечним світами. Він завжди перебуває у захищеному стані.
- S-EL0 - Secure EL0, рівень, на якому запускаються і працюють довірені застосунки.
- S-EL1 - Secure EL1, рівень для виконання довіреної операційної системи, або ж TrustedOS.

Як уже було зазначенно вище по тексту контоль і взаємодія між світами здійснюється за допомогою Secure Monitor Call. Виклики генеруються за допомогою SMC і обробляються на рівні Secure Monitor.

Рис. 1.9 – Рівні виконання

1.8 Trusted Execution Environment

Довірене середовище виконання (TEE) - це середовище, де виконуваний код та дані, до яких здійснюється доступ, є ізольованими та захищеними з точки зору конфіденційності (ніхто не має доступу до даних) та цілісності (ніхто не може змінити код та його поведінку).

Довірене середовище виконання, включаючи смартфони, телевізійні приставки, відеоігри та консолі Smart TV.

Важливо розуміти для чого нам необхідне безпечне середовище виконання. Програмне забезпечення стає дедалі складнішим. Більші проекти, такі як ядро Linux, мають мільйони рядків коду. А це означає багато помилок.

Ми виправляємо помилки і іноді виявляємо регресії. Ми додаємо нові функції, а разом з ними і деякі помилки. Хоча ми можемо працювати над тим, щоб запобігти деяким типам помилок, ми завжди матимемо помилки в програмному забезпеченні. І деякі з цих помилок можуть виявити вразливість системи безпеки. Гірше того, якщо помилка знаходиться в ядрі, вся система скомпрометована.

Одним із способів вирішення цієї проблеми є створення ізольованого середовища, де, навіть якщо операційна система ушкоджена, ваші дані захищені. Це те, що ми називаємо довіреним середовищем виконання або TEE.

І існує кілька випадків використання TEE[6]:

- TEE буде хорошим рішенням для зберігання та управління ключами шифрування пристрою, які можна використовувати для перевірки цілісності операційної системи.
- TEE добре підходить для впровадження методів біометричної автентифікації (розпізнавання обличчя, датчика відбитків пальців та голосової авторизації), ізоляції ресурсів у пристрої для зберігання біометричного алгоритму, облікових даних користувача та пов’язаних даних.
- TEE можна використовувати в мобільних додатках для електронної комерції, таких як мобільні гаманці, однорангові платежі або безконтактні платежі для зберігання та управління обліковими даними та конфіденційними даними.
- TEE також є підходящим середовищем для захисту цифрової інформації, захищеної авторським правом (книги, фільми, аудіо тощо) на підключених пристроях, таких як смартфони, планшети та смарт-телевізори. Незважаючи на те, що цифровий вміст захищений під час передачі або потокового передавання за допомогою шифрування, TEE захищатиме вміст після того, як його буде розшифровано на пристрої, гарантуючи, що розшифрований вміст не піддається впливу середовища операційної системи.

У системі з TEE у нас є ненадійні програми, що працюють у середовищі багатофункціонального виконання (REE), і надійні програми (TA), що працюють у середовищі Trusted Execution Environment (TEE).

Тільки надійні програми, що працюють на TEE (Secure World), мають повний доступ до основного процесора, периферійних пристроїв та пам'яті, тоді як апаратна ізоляція захищає їх від ненадійних програм, що працюють в основній операційній системі (Non-Secure World).

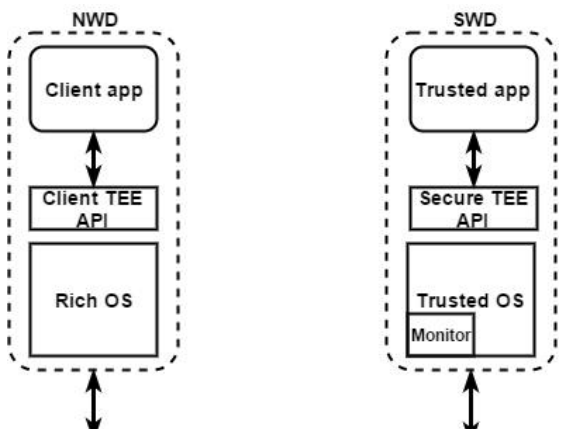

Рисунок 1.10 - Емуляція TEE

Хоча на наведеній вище схемі зображено TEE з операційною системою (Trusted OS), ми могли б мати просто оголену прошивку, що виставляє інтерфейс з ексклюзивним доступом до певних апаратних ресурсів.

У TEE всі надійні програми (TA) та пов'язані з ними дані повністю ізольовані від звичайної (ненадійної) операційної системи та їх програм. Крім того, надійні програми повинні працювати ізольовано від інших надійних програм та від самого TEE.

Крім того, TEE приймає до виконання лише той код, який був належним чином санкціонований та перевірений іншим уповноваженим кодом. Тож у TEE нам потрібна захищена функція завантаження для перевірки цілісності та справжності всіх компонентів операційної системи (завантажувачів, ядра, файлових систем, надійних програм тощо). Це гарантує, що ніхто не втручався в код операційної системи, коли пристрій було вимкнено.

TEE - це справді середовище виконання (з операційною системою або без неї), що має ексклюзивний доступ до певних апаратних ресурсів.

Висновки до розділу 1

В даному розділі було розглянуто операційну систему Linux та основні функції як операційної ситеми загалом, так і зокрема функції Linux. Основним

завданням операційної системи є створення певного інтерфейсу між користувачем комп'ютера та апаратними засобами такими, як процесор, оперативна пам'ять, різного роду переферійні пристрої, такі як принтери, монітори, миші та інше.

Основними функціями операційної системи є:

- Багатозадачність. У операційній системі, що називається багатозадачною, мають бути реалізовані певні базові механізми такі, як система ахисту пам'яті, обробник переривань, системний таймер та планувальник системного часу процесора. У багатозадічній операційній системи можна запускати відразу декілька процесів, навіть за усови наявності лише одного фізичного ядра.
- Контроль пам’яті. Операційна система бере на себе роботу із розподілу наявної в системі пам'яті між процесами, які її потребають.
- I\O. Операційна система відповідає, а обробку і контроль усіх каналів вводу та виводу для того, що не було безконтрольного та конфліктого доступу до цих резурсів зі сторони зацікаавлених процесів.
- Міжпроцесна взаємодія.
- та інші.

Ядро Linux є важливою частиною Linux-подібної операційної системи. Воно підтримує динамічне завантаження модулів ядра, які можуть розширювати доступний фунціонал ядра Linux.

В операційній системі існують механізми захисту пам’яті. Вони ускладнюють експлуатацію вразливостей (buffer overflow, integer overflow тощо).

Було розглянуто технологію Trusted Execution Environment. Її завданняполягає у тому, щоб розділи простори виконання процесів та операційних систем на два серидовища, так звані безпечне та небезпечне середовище виконання. Цей підхід дозволяє зберегти конфіденційні дані неушкодженими у вападку помилок у ядрі операційної системи або ж компрометаціхзахисту системи.

2 ОГЛЯД ЗАГРОЗ ТА АНАЛІЗ ІСНУЮЧИХ МЕХАНІЗМІВ ЗАХИСТУ

2.1 Типові вразливості та методи захисту в user mode

Для операційній системи Linux властиві звичайні загрози інформаційній безпеці, що пов'язані із недосконалістю програмноо забезпечення або ж програмногозабезпечення, що створено якраз для зловмисних дій у операційній системі. Для прикладу ми можемо мати в нашому застосунку щось накшталт переповнення масиву (рис. 2.1).

```c
#include <stdio.h>
#include <stdlib.h>

void never_execute()
{
    printf("It should never be executed");
    return;
}
```

Рис. 2.1 – Переповнення масиву

На рисунку 2.1 зображено тривіаьний приклад переповнення масиву, ця помилка може бути допущення навіть по неуважності програміста. Ми бачимо 3 функції, а саме main, головну функцю, з якої починає виконання наш застосунок, також ми бачемо, що наша основна функція main() приймаэ два параметра із командного рядка запуску.Також ми маємо функцію vulnerable(), що приймає аргумент із командного рядка перенаправлений функцією main і копіює його у виділений бефер, що має константний розмір 10. Останньою функцією є never_execute(), яка начебто ніколи не має бути виконана, оскільки у програмному коді відсутні прямі виклики цієї функції. Обмежуючи буфер ми спричиняємо вразливість переповнення, оскільки користувач, що не знає про особливості нашого програмного коду може передати в командний рядок запуску, щось що займає більше десяти символів.

Візьмемо до уваги приклад, що наведений на рис. 2.1, після компіляї і запуску нашого застосунку дані будуть записанні у різні облаі пам'яті. Ми коротко розповідали про це у першому розділі, тепер прийшов час розглянути це питання більш детально.

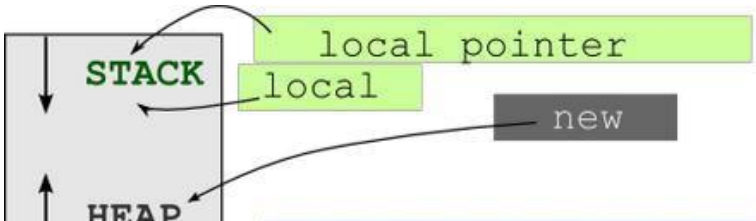

Рис. 2.2 – сегменти пам’яті

- Stack – це такий сегмент даних, у якому містяться локальні дані, адреса функцій та аргументи, з якими їх викликають.
- Heap - це такий сегмент пам'яті, у якому розміщуються динамічно виділені блоки пам'яті.
- Text або Code segment – це сегмент, у якому безпосередньо розміщується бінарний код застосунку, який починає виконуатись при запуску нашого застосунка.

- Data – цей блок містить ініцалізовані глобальні змінні.

- BSS – сегмент, що містить дані, які не були ініціалізовані.

У даному випадку нас цікавить стек і функція vulnerable, що має локальну змінну buf розміром 10 байтів, що розміщуються на стеку після чуго виконується функція strcpy().

```
void vulnerable(char *in)
{
    char buf[10];
    strcpy(buf, in);
}
```

```
vulnerable:
  stmfd sp!, {fp, lr}
  add fp, sp, #4
  sub sp, sp, #24
  str r0, [fp, #-24]
  sub r3, fp, #16
  mov r0, r3
  ldr r1, [fp, #-24]
  bl strcpy
  sub sp, fp, #4
  ldmfd sp!, {fp, pc}
```

Рис. 2.3 – Функція vulnerable (ARM)

Як уже зазначалось, якщо зловмисник подасть на вхіднашій проорамі дані, що а розміром перевищують десять байт, винекне так званий buffer overflow.

За допомогою цієї вразливості зловмисник може отримати доступ до регістру LR і заволодіти зворотнім адресом, що дасть зловмиснику можливість повністю контролювати програму і передати управління, для прикладу, своєму кодові або ж функції never_execute(). Для прикладу:

```
main:
  stmfd sp!, {fp, lr}
  add fp, sp, #4
  sub sp, sp, #8
  str r0, [fp, #-8]
  str r1, [fp, #-12]
  ldr r3, [fp, #-12]
  add r3, r3, #4
  ldr r3, [r3, #0]
  mov r0, r3
  bl vulnerable
  mov r3, #0
  mov r0, r3
  sub sp, fp, #4
  ldmfd sp!, {fp, pc}
```

Рис. 2.4 – Наша функцыя main() на асемблері (ARM)

Тепер запустимо наш застосунок у додатку для дебагу під назвою gdb у емуляторі віртуальної машини ARM під назвою QEMU, та оглянемо код наших функцій на асемблері (рис. 2.5 – 2.7):

```
Dump of assembler code for function main:
=> 0x0000845c <+0>:     push    {r11, lr}
   0x00008460 <+4>:     add     r11, sp, #4
   0x00008464 <+8>:     sub     sp, sp, #8
   0x00008468 <+12>:    str     r0, [r11, #-8]
   0x0000846c <+16>:    str     r1, [r11, #-12]
   0x00008470 <+20>:    ldr     r3, [r11, #-12]
   0x00008474 <+24>:    add     r3, r3, #4
   0x00008478 <+28>:    ldr     r3, [r3]
   0x0000847c <+32>:    mov     r0, r3
   0x00008480 <+36>:    bl      0x842c <vulnerable>
   0x00008484 <+40>:    mov     r3, #0
   0x00008488 <+44>:    mov     r0, r3
   0x0000848c <+48>:    sub     sp, r11, #4
   0x00008490 <+52>:    pop     {r11, lr}
   0x00008494 <+56>:    bx      lr
End of assembler dump.
```

Рис. 2.5 – main() на асемблері

```
Dump of assembler code for function vulnerable:
   0x0000842c <+0>:     push    {r11, lr}
   0x00008430 <+4>:     add     r11, sp, #4
   0x00008434 <+8>:     sub     sp, sp, #24
   0x00008438 <+12>:    str     r0, [r11, #-24]
   0x0000843c <+16>:    ldr     r3, [r11, #-24]
   0x00008440 <+20>:    sub     r2, r11, #16
   0x00008444 <+24>:    mov     r0, r2
   0x00008448 <+28>:    mov     r1, r3
   0x0000844c <+32>:    bl      0x8320 <strcpy>
   0x00008450 <+36>:    sub     sp, r11, #4
   0x00008454 <+40>:    pop     {r11, lr}
   0x00008458 <+44>:    bx      lr
End of assembler dump.
```

Рис. 2.6 – vulnerable() на асемблері

```
Dump of assembler code for function never_execute:
   0x00008408 <+0>:     push    {r11, lr}
   0x0000840c <+4>:     add     r11, sp, #4
   0x00008410 <+8>:     ldr     r3, [pc, #16]   ; 0x8428 <never_execute+32>
   0x00008414 <+12>:    mov     r0, r3
   0x00008418 <+16>:    bl      0x8314 <printf>
   0x0000841c <+20>:    sub     sp, r11, #4
   0x00008420 <+24>:    pop     {r11, lr}
   0x00008424 <+28>:    bx      lr
   0x00008428 <+32>:    andeq   r8, r0, r12, lsl r5
End of assembler dump.
```

Рис. 2.7 – never_execute() на асемблері

Як можна побачити, герістри r0, r1 будуть використовуватись для передачі параметрів у функцію strcpy().

```
Dump of assembler code for function vulnerable:
   0x0000842c <+0>:     push    {r11, lr}
   0x00008430 <+4>:     add     r11, sp, #4
   0x00008434 <+8>:     sub     sp, sp, #24
   0x00008438 <+12>:    str     r0, [r11, #-24]
   0x0000843c <+16>:    ldr     r3, [r11, #-24]
   0x00008440 <+20>:    sub     r2, r11, #16
=> 0x00008444 <+24>:    mov     r0, r2
   0x00008448 <+28>:    mov     r1, r3
   0x0000844c <+32>:    bl      0x8320 <strcpy>
```

Рис. 2.8 – vulnerable() та регістри r2, r3

Після загрузки регістрів r0, r1 викликається функція strcpy. Перехід у цю функцію можна побачити на рис. 2.9.

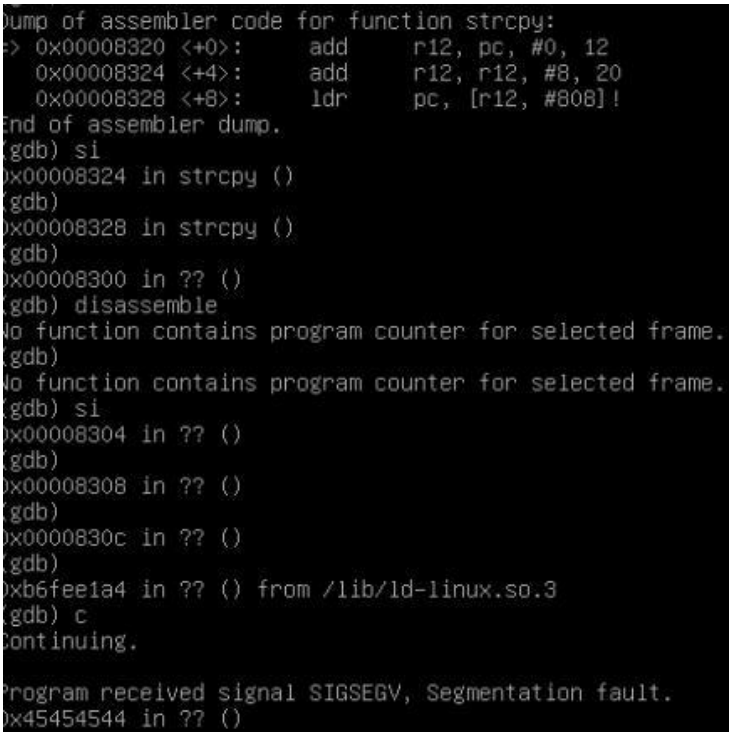

Рис. 2.9 – Перетирання адреси повернення

А після цього виконання знову повератється до функції vulnerable. Оскільки адреса повернення була перезаписана, то виконання перейде у невідоме місце і якщо ми замінимо цей адрес на адрес функції never_execute , то ми перейдемо саме у неї (рисунок 2.10).

Рисунок 2.10 – Виконання функції never_execute

Це простий приклад, який зрештою ніякої шкоди системі не несе, але на базі цього можна зробити значно небезпечніші застосунки, які для прикладу можуть запросити права суперкористувача і робити із операційною системою все, що заманеться, без нашого відому.

Одним із найпростіших методів захисту від даного виду вразливостей є компіляція нашого застосунку і з спеціальними флагами компіляції, а саме із прапорцем stack_protector, проте варто зазначити, що назва цього прапорця на різних компіляторах може відрізнятися.

2.3 Огляд SELinux

Сьогодні Linux використовується на мільйонах смартфонів, серверів та звичайних комп'ютерів, тож абсолютно не дивно, що ядро Linux є об'єктом пильного спостереження в контексті кібербезпеки для багатьох програмістів. Як відомо Linux, це продукт із відкритим програмним кодом і будь-то може стати розробником або ж, як мінімум, вказати на певні недоліки ядра. Такий підхід до розробки програмного забезпечення приніс свої плоди, зокрема було розроблено механізм SELinux.

SELinux (SELinux)[7] - це система примусового контролю доступу, реалізована на рівні ядра. Вперше ця система з'явилася в четвертій версії CentOS, а в 5 і 6 версії реалізація була істотно доповнена і покращена. Ці поліпшення дозволили SELinux стати універсальною системою, здатною ефективно вирішувати масу актуальних завдань. Варто пам'ятати, що класична система прав Unix застосовується першою, і управління перейде до SELinux тільки в тому випадку, якщо ця первинна перевірка буде успішно пройдена.

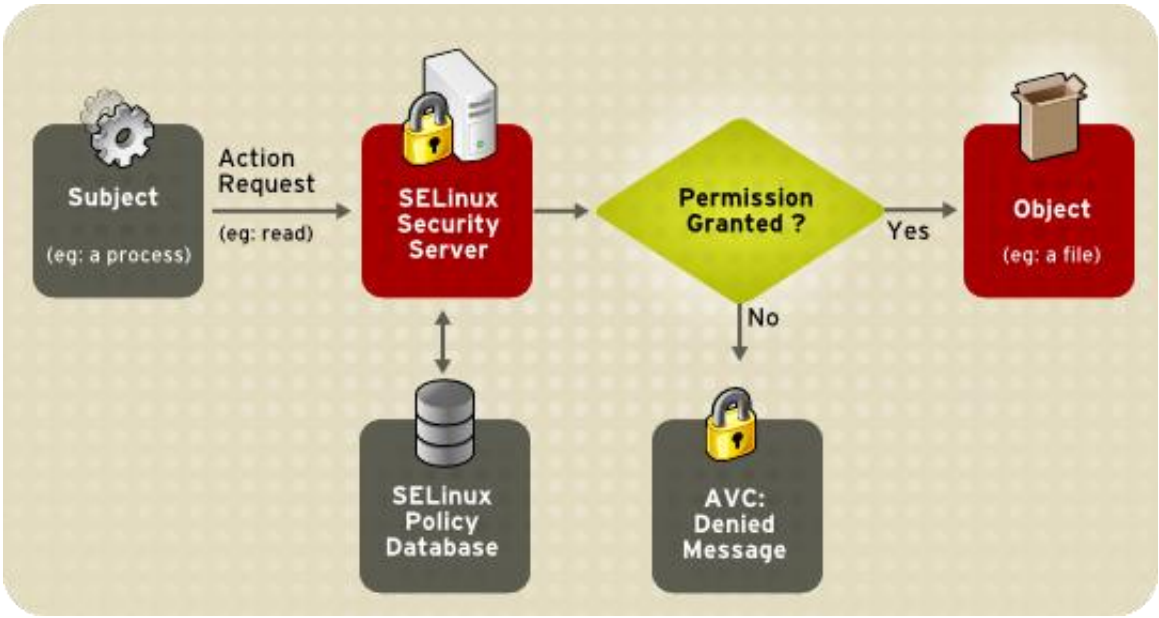

Рисунок 2.11 – Принцип прийняття рішень у SELinux

Для того, щоб зрозуміти, в чому полягає практична цінність SELinux, розглянемо кілька прикладів, коли стандартна система контролю доступу недостатня. Якщо SELinux відключений, то вам доступна тільки класична дискреційна система контролю доступу, яка включає в себе DAC (виборче управління доступом) або ACL (списки контролю доступу). Тобто мова йде про маніпулювання правами на запис, читання і виконання на рівні користувачів і груп користувачів, чого в деяких випадках може бути абсолютно недостатньо. наприклад:

- Адміністратор не може в повній мірі контролювати дії користувача. Наприклад, користувач цілком здатний дати всім іншим користувачам права на читання власних конфіденційних файлів, таких як ключі SSH.

- Процеси можуть змінювати налаштування безпеки. Наприклад, файли, що містять в собі пошту користувача повинні бути доступні для читання тільки одному конкретному користувачеві, але поштовий клієнт цілком може змінити права доступу так, що ці файли будуть доступні для читання всім.

- Процеси успадковують права користувача, який їх запустив. Наприклад, заражена трояном версія браузера Firefox в стані читати SSH-ключі користувача, хоча не має для того жодних підстав.

По суті, в традиційній моделі виборчого керування доступом (DAC), добре реалізовані тільки два рівня доступу - користувач і привілейований користувач. Немає простого методу, який дозволив би встановлювати для кожного користувача необхідний мінімум привілеїв.

Звичайно, є безліч методів обходу цих проблем в рамках класичної моделі безпеки, але жоден з них не є універсальним.

SELinux відповідає моделі мінімально необхідних привілеїв для кожного сервісу, користувача і програми набагато більш строго. За замовчуванням встановлено «заборонний режим», коли кожен елемент системи має тільки ті права, які життєво необхідні йому для функціонування. Якщо ж користувач, програма або сервіс намагаються змінити файл або отримати доступ до ресурсу, який явно не потрібен для вирішення їх, то їм буде просто відмовлено в доступі, а така спроба буде зареєстрована в журналі.

SELinux реалізована на рівні ядра, так що прикладні програми можуть зовсім нічого не знати про версії цієї системи примусового контролю доступу, особливості її роботи і т.д. У разі грамотної настройки, SELinux ніяк не вплине на функціонування сторонніх програм і сервісів. Хоча, якщо додаток здатний перехоплювати повідомлення про помилки цієї системи контролю доступу, зручність користування таким додатком істотно зростає. Адже в разі спроби отримати доступ до захищеного ресурсу або файлу, SELinux передає в основне додаток помилку з сімейства «access denied». Але лише деякі додатки використовують одержувані від SELinux коди повернення системних викликів.

Ось кілька прикладів використання SELinux, які дозволяють побачити, яким чином можна збільшити ступінь безпеки всієї системи.

- Створення та налагодження списку програм, які можуть читати ssh-ключі.
- Запобігання несанкціонованого доступу до даних через mail-клієнт.
- Налаштування браузера таким чином, щоб він міг читати в домашній папки користувача тільки необхідні для функціонування файли і папки.

SELinux має три основних режим роботи, при цьому за умовчанням встановлено режим Enforcing. Це досить жорсткий режим, і в разі необхідності він може бути змінений на більш зручний для кінцевого користувача.

- Enforcing: Режим за замовчуванням. При виборі цього режиму всі дії, які якимось чином порушують поточну політику безпеки, будуть блокуватися, а спроба порушення буде зафіксована в журналі.

- Permissive: У разі використання цього режиму, інформація про всі дії, які порушують поточну політику безпеки, будуть зафіксовані в журналі, але самі дії не будуть заблоковані.

- Disabled: Повне відключення системи примусового контролю доступу.

Ви можете подивитися поточний режим і інші настройки SELinux (а в разі необхідності і змінити його) за допомогою спеціального GUI-інструменту, доступного в меню «Адміністрування» (system-config-selinux). Якщо ж ви звикли працювати в консолі, то можете подивитися поточний статус командою sestatus.

 # sestatus

SELinux status: enabled

SELinuxfs mount: /selinux

Current mode: enforcing

Mode from config file: enforcing

Policy version: 21

Policy from config file: targeted

Також ви можете дізнатися статус SELinux за допомогою команди getenforce.

Команда «setenforce» дозволяє швидко перемикатися між режимами Enforcing і Permissive, зміни вступають в силу без перезавантаження. Але якщо ви включаєте

або відключаєте SELinux, потрібне перезавантаження, адже потрібно заново встановлювати мітки безпеки в файлової системі.

Для того, щоб вибрати режим за замовчуванням, який буде застосовуватися при кожному завантаженні системи, варто задати значення рядка 'SELINUX =' у файлі /etc/selinux/config, задавши один з режимів - 'enforcing', 'permissive' або 'disabled' . Наприклад: 'SELINUX = permissive'.

Як зазначалося раніше, SELinux за замовчуванням працює в режимі Enforcing, коли будь-які дії, крім дозволених, автоматично блокуються, кожна програма, користувач або сервіс володіють тільки тими привілеями, які необхідні їм для функціонування, але не більше того. Це досить жорстка політика, яка володіє як плюсами - найбільший рівень інформаційної безпеки, так і мінусами - конфігурація системи в такому режимі пов'язане з великими трудовитратами системних адміністраторів, до того ж, великий ризик того, що користувачі зіткнуться з обмеженням доступу, якщо захочуть використовувати систему хоч скільки-небудь нетривіальним чином. Такий підхід допустимо в Enterprise-секторі, але неприйнятний на комп'ютерах кінцевих користувачів. Багато адміністраторів просто відключають SELinux на робочих станціях, щоб не стикатися з подібними проблемами.

Для того, щоб уникнути цього, для ключових додатків і сервісів, таких як, наприклад, httpd, named, dhcpd, mysqld, визначені заздалегідь сконфігуровані цільові політики, які не дозволять отримати зловмисникові доступ до важливих даних. Ті ж додатки, для яких політика не визначена, виконуються в домені unconfined_t і не захищаються SELinux. Таким чином, правильно вибрані цільові політики дозволяють домогтися прийнятного рівня безпеки, не створивши при цьому для користувача зайвих проблем.

SELinux надає наступні моделі управління доступом:

- Type Enforcement (TE): основний механізм контролю доступу, який використовується в цільових політиках. Дозволяє детально, на найнижчому рівні управляти дозволами. Найгнучкіший, але і самий трудомісткий для системного адміністратора механізм.

- Role-Based Access Control (RBAC): в цій моделі права доступу реалізуються в якості ролей. Роллю називається дозвіл на виконання певних дій одним або декількома елементами системи над іншими частинами системи. По-суті, RBAC є подальшим розвитком TE.

- Multi-Level Security (MLS): багаторівнева модель безпеки, в якій всім об'єктам системи присвоюється певний рівень доступу. Дозвіл або заборона доступу визначається тільки співвідношенням цих рівнів.

  Всі процеси і файли в рамках SELinux мають контекст безпеки.

2.4 Огляд Руткітів

Руткіт (rootkit) - набір утиліт для приховування слідів перебування кіберзлочинців на зараженій машині, активності шкідливих програм і небажаних або потенційно небезпечних програм: сніфферов, кейлогерів, засобів віддаленого адміністрування і т.д. Багато руткітів дозволяють зломиснику встановлювати в систему нові програми і виконувати інші різноманітні маніпуляції. Заразивши безліч комп'ютерів руткитами, зловмисник може проводити з них спам- або DDoS-атаки[8].

Хакер встановлює руткіт в систему після отримання привілеїв адміністратора. Руткіт (rootkit) може маскувати шкідливі програми під стандартні системні утиліти,

збирати інформацію про систему і управляти процесами в ній. Способи впровадження та функціонування руткита залежать від цільової платформи.

За типом привілеїв руткіти діляться на два види: призначений для користувача (виконує дії від імені облікового запису користувача) і працює на рівні ядра. Принцип роботи руткітів теж можна розділити на дві категорії: що змінюють алгоритми системних функцій і змінюють системні структури даних. В операційних системах руткіти можуть працювати наступним чином[15]:

1. Захоплення таблиць викликів. Такий руткит (rootkit) здатний діяти як на рівні користувача, так і на рівні ядра. Змінюючи таблицю, він перенаправляє виклики системних функцій на потрібні йому адреси - наприклад, на ті, за якими знаходяться функції троянської програми. В результаті перехоплена процедура може обійти антивірус (шляхом блокування викликів від нього) або замінити вихідну функцію. Є підстави вважати даний тип руткітів найвитонченішим. Його можливості ширші, ніж у другого типу, описаного далі. Викликано це тим, що руткіти-загарбники таблиць викликів можуть працювати як на рівні ядра, так і на рівні користувача.

2. Модифікація коду функції. При даному виді роботи руткита перші кілька байт функції, що викликається замінюються на шкідливий код. Реалізація цього підходу має нюанс: для кожного виклику перехопленої функції потрібно спочатку відновлювати її машинний код до стану, в якому він був до виклику, щоб потім знову перехопити її. Перехоплювач працює за наступним алгоритмом:

- Виконує дії, які задумав зловмисник.
- Відновлює перші байти перехопленої функції.
- Аналізує вихідні дані функції.
- Повертає управління функцією системі.

Також для перехоплення функції можна замінити її перші байти операцією jmp, яка передає управління руткіта. Однак таку операцію легко викрити, якщо перевіряти перші байти що викликаються функцій на її наявність, тому більшість

кіберзлочинців «затирає» кілька байт перед операцією jmp безглуздими операціями на кшталт mov a, b.

Для UNIX-подібних операційних систем руткіти можуть реалізовуватися наступним чином:

- Заміною системних утиліт.
- У вигляді модуля ядра.
- За допомогою патчінга VFS.
- Перехопленням таблиць системних викликів.
- Зміною фізичної пам'яті ядра.

Установка руткита - перше, що робить зломщик, коли отримує права адміністратора в системі. Крім виконання коду, руткіти можуть додавати в систему свої, «невидимі» служби і драйвери, ховати будь-які директорії і файли. Основне призначення руткита - потайне знаходження в операційній системі, і він з ним прекрасно справляється. Руткіти можуть використовуватися правоохоронними органами для збору відомостей про зловмисника, доступу до його ПК і до конфіденційної інформації, імовірно містить докази. Без спеціального програмного забезпечення знайти руткіти в системі майже неможливо. Це - ще одна їх перевага.

Багато дій руткита в системі можна виявити. Так, підміна системних утиліт легко виявляється засобами на кшталт SELinux. Також існують спеціалізовані програми-антіруткіта, які мають вдосконалені методи виявлення і видалення таких шкідливих об'єктів з системи. Ряд антивірусів для робочих станцій має в своєму складі повноцінний антіруткит-модуль. Руткітом може бути заражений і сервер великої компанії, і приватний комп'ютер. Багато руткітів мають можливості, що мона співставити з функціональністю засобів віддаленого адміністрування (RAT). Зараз у вільному доступі є програми-конструктори, за допомогою яких навіть недосвідчений хуліган може «побудувати» свій руткит і заразити їм інформаційну систему або чийсь комп'ютер. Уразливості в браузерах дозволяють запустити

процес впровадження руткита на комп'ютері жертви, навіть якщо вона не скачувала ніяких файлів, а просто зайшла на заражений сайт.

Давайте розглянемо приклад із заміною таблиці системних викликів. На рисунку 2.11, можна побачити код модулю ядра, який буде мінювати таблицю системних викликів, так щоб викликалась потрібна нам функція.

```
asmlinkage long (*real_chdir)(const char __user *filename);

unsigned long *syscall_table = (unsigned long*)0xffffffff9e600240;

asmlinkage long chdir_patch(const char __user *filename)
{
    printk("Patchded chdir is executed!\n");
    return (*real_chdir)(filename);
}

int __init chdir_init(void)
{
    unsigned int l;
    pte_t *pte;
    pte = lookup_address((unsigned long)syscall_table, &l);
    pte->pte |= _PAGE_RW;
    real_chdir = (void*)*(syscall_table + __NR_chdir);
    *(syscall_table + __NR_chdir) = (unsigned long)chdir_patch;
    printk("Patched!\nOLD :%p\nIN-TABLE:%p\nNEW:%p\n", real_chdir, syscall_table[__NR_chdir], chdir_patch);
    return 0;
}

void __exit chdir_cleanup(void)
{
    unsigned int l;
    pte_t *pte;
    *(syscall_table + __NR_chdir) = (unsigned long)real_chdir;
    pte = lookup_address((unsigned long)syscall_table, &l);
    pte->pte &= ~_PAGE_RW;
    printk("Cleanup,exit\n");
    return 0;
}

module_init(chdir_init);
module_exit(chdir_cleanup);
MODULE_LICENSE("GPL");
```

Рис. 2.12 – Приклад руткіта

Як можна помітити, вище описаний руткіт створено у вигляді завантажуаного модуля ядра. Такий висновок можна зробити помітивши функції module_init та module_exit. Функція module_init використовується одного разу при ініціалізації модулю, так само функія module_exit виконується одного разу при вивантаженні модуля із ядра. Нагадаємо, що за завантаження модолю у ядра відповідає системна команда Linux insmod, а за вивантаження модуля - команда rmmod.

У 16-му рядку на рис. 2.10 можна помітити оголошення змінної syscall_table. Значення адреси таблиці системних викликів було знайдено за допомогою команди -

cat /proc/kallsyms | grep sys_call_table (рисунок 2.13)

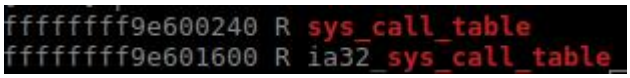

Рис. 2.13 – Результат виклику наведеної команди

На рисунку 2.13 можна помітити доволі простий метод визначення адресу таблиці системних викликів, а також прав доступу до неї.

Для різного роду процесів ця таблиця, як можна помітити, доступна лише у режимі читання, але це не так вже й важко змінити. 18 рядок із наведеного вище прикладу визначає новий системний виклик, у нашому випадку це функція chdir_patch. Не зважаючи на виклик основноъ функцыъ можна помытити, що так само буде викликано функцію printk, яка запише відповідне повідомлення у системний лог. І саме по факту наявності цього повідомлення у системному лозі ми зрозуміємо, що підміна вдалась.

Далі у рядку 24 ми маєму функцію ініціалізації нашої підміни. У цій функції можна знайти імлементовану логіку зміни прав доступу до таблиці системних викликів. У рядку 31 ми підміняємо реальної функції на нашу модифіковану функцію.

```
*(syscall_table + __NR_chdir) = (unsigned long)chdir_patch;
```

Рис. 2.14 – Підміна функції.

__NR_chdir – це відступ, ми використовуєм, щоб дізнатися реальний адрес необхідної нам функції.

Рядок 36 описує логіку вивантаження нашого мудуля із ядра. У ній ми встановлюємо попередні значення у таблицю системних викликів.

Після завантаження модулю і подальших викликів команди cd та rmmod, в логах dmesg можна побачити наступне (рисунок 2.24 - 2.26):

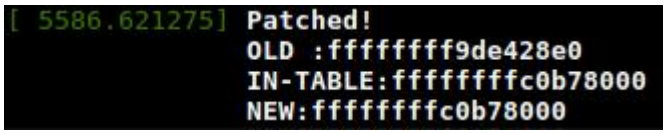

Рис. 2.15 – Системні логи після встановлення модулю

Рис. 2.16 – Системні логи після виконання команди cd

Рис. 2.17 – Системні логи після вивантаження модулю

Таким чином, ми переконалися у тому, що маючи необхідні права зловмисники мають змогу створювати та завантажувати у ядро операційної системи Linux шідливе програмне забезпечення, яке може змінювати таблицю системних викликів і таим чином передавати виконання коду у місця абчолютно невідомі кінцевому користувачу машини. Звісно є аналізатори руткітів, проте процес виявлення складніших руткітів не є тривіальною задачею.

Всі сучасні версії руткітів можуть ховати від користувача файли, папки і параметри реєстру, приховувати програми, системні служби, драйвери і мережеві з'єднання.

Одним з найбільш небезпечних руткітів є екземпляр з назвою «FU», виконаний частково як додаток, а частково як драйвер. Він не займається перехопленнями, а маніпулює об'єктами ядра системи, тому знайти такого шкідника дуже складно. Якщо ви виявили руткит, це ще не означає, що ви зможете позбутися від нього. Для захисту від знищення користувачем або антивірусом в руткіти застосовується декілька технологій, які вже зустрічаються і в шкідливих програмах інших типів. Наприклад, запускаються два процеси, які контролюють один одного. Якщо один з них припиняє роботу, другий відновлює його. Застосовується також схожий метод, який використовує потоки: віддалений файл, параметр реєстру або знищений процес через деякий час відновлюються.

Популярний спосіб блокування доступу до файлу: файл відкривається в режимі монопольного доступу або блокується за допомогою спеціальної функції; видалити такий файл стандартними способами неможливо. Якщо спробувати скористатися відкладеним видаленням (під час наступного завантаження), то, швидше за все, запис про цю операцію буде через деякий час видалено або файл буде перейменовано.

На жаль, існуючі на сьогоднішній день антивірусні програми, призначені для виявлення всіляких вірусів і руткітів, не дають стовідсоткової гарантії безпеки. Володіючи вихідним кодом цих програм, можна створити будь-які модифікації руткітів або включити частину коду в будь-яку шпигунську програму. Головна відмінність руткітів це здатність не міцно закріпитися в системі, а проникнути в неї.

2.5 Аналіз задачі виявлення руткітів.

Незважаючи на витончені способи маскування руткітів, встановити, що система заражена, часто не так складно. Руткіти перехоплюють функції, а тут, як показано раніше, варіантів небагато.

У випадках з підміною адрес найкраще мати вихідний варіант таблиці, щоб згодом можна було перевірити її цілісність, але потрібно пам'ятати, що після поновлення ядра адреси можуть змінитися. Для сплайсингу можна перевірити, чи немає за адресою функції байта зі значенням 0x90, відповідного jmp, і виявити хук (що вже не настільки просто, якщо він встановлений не на початку).

Виявити приховані підключення в найпростішої ситуації можна зовнішнім сканером портів, але при використанні Port knocking це навряд чи дасть результат, хіба що доведеться моніторити трафік, що проходить через мережевий шлюз.

Ще одним із методів є пошук в пам'яті ядра дескриптора модуля, який знаходиться у відірваному від списку завантажених модулів стані. Знайшовши, можна повернути його в список, щоб потім використовувати rmmod (якщо, звичайно, руткит не підміняючи і його про всяк випадок). Існує рішення, що дозволяє виявити в пам'яті об'єкти, схожі на дескриптори LKM, але воно працює на вже дуже старих і тільки 32-бітових ядрах. Перебір же пам'яті сучасних систем ускладнений через величезну адресного простору[10].

Однак встановити наявність руткита в системі - лише півсправи. Непогано б його звідти прибрати, а ще й знайти сам виконуваний файл, щоб було що повивчати на вихідних. Найпростіший і безвідмовний спосіб - взяти носій із зараженою операционкой і шукати там підозрілі файли, що завантажуються модулів зі сторонньою машини. До того ж, напевно існують системи, для яких перебувати в неробочому стані, поки йде пошук, вкрай небажано. Тому спробуємо дослідити заражену машину зсередини. Що ж може видати присутність руткіта і при цьому вказати на файл шкідливого модуля?

Що LKM-руткіт неодмінно повинен десь прописати рядок, що веде до його завантаженні при старті ОС. Також ми знаємо, що він намагається це приховати.

Припустимо, в нашій системі встановлений Reptile. Відкриваємо в будь-якому текстовому редакторі /etc/modules і бачимо, що він нібито чистий, але ми-то знаємо - там є приховане вміст. Що буде, якщо спробувати, не вносячи жодних змін в файл, зберегти його? Чи збережеться саме те, що ми бачимо, - тобто після перезавантаження ОС шкідливий LKM вже активний не буде. Описаний ефект спостерігається тому, що зберігається рівно те, що відкрито в редакторі, тобто частина файлу, вільна від даних руткіта.

Але що робити, якщо невідомо, в якій саме файл автозавантаження прописався руткіт? Чи не перебирати ж всі задіяні. До того ж такий спосіб знищує в файлі все зачіпки, які допомогли б швидше ідентифікувати шкідливий руткіт. Потрібно знайти модифікований файл автозавантаження, помістити замість нього вільну від даних руткита копію, але при цьому якось залишити ці дані, щоб було простіше знайти шкідливий LKM[9]. Ще було б корисно визначити, який саме файл був модифікований руткітом, щоб не пересохранять і аналізувати їх все.

Таким чином, наша задача запобігання завантаження LKM-руткита зводиться до двох питань:

- як знайти файл із прихованим вмістом, якщо невідомо, яким чином руткіт закріпився в системі;
- як зберегти цей файл так, щоб цей вміст не виявився видаленим.

2.6 Огляд існуючих рішень

Для нашої роботи буде важливо розглянути такі рішення, як kprobes та jprobes[11].

Probes - це механізм налагодження ядра Linux, який також може використовуватися для моніторингу подій у системі. Ви можете використовувати його для усунення вузьких місць із низькою продуктивністю, реєстрації

конкретних подій, проблем з трасуванням тощо. KProbes був розроблений IBM як основний механізм для іншого інструменту відстеження вищого рівня, який називається DProbes. DProbes додає ряд функцій, включаючи власну мову сценаріїв для написання обробників зондів. Однак лише KProbes були включені у стандартне ядро Linux.

Зонд ядра - це набір обробників, розміщених за певною адресою інструкцій. На даний момент у ядрі є два типи зондів, які називаються "KProbes" і "JProbes". KProbe визначається попереднім обробником. Коли KProbe встановлюється за певною інструкцією і ця інструкція виконується, попередній обробник виконується безпосередньо перед виконанням зондованої інструкції. Подібним чином, обробник повідомлення виконується відразу після виконання зондованої інструкції. JProbes використовуються для отримання доступу до аргументів функції ядра під час виконання. JProbe визначається обробником JProbe з тим самим прототипом, що і у функції, аргументи якої мають бути доступні. Коли виконується зондована функція, керування спочатку передається визначеному користувачем обробнику JProbe, а потім передається виконання у вихідну функцію. Пакет KProbes розроблений таким чином, що інструменти для налагодження, трасування та ведення журналу можуть бути створені шляхом його розширення.

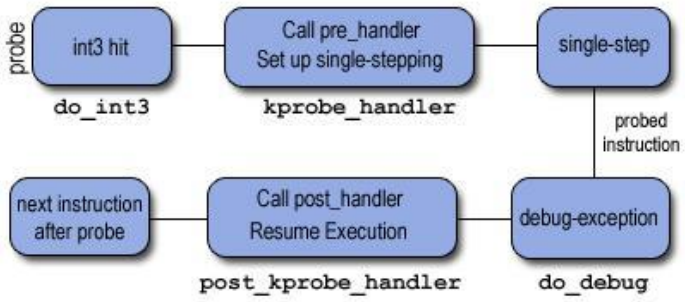

Рис. 2.18 – Принцип kprobes

Вище зазначений малюнок описує архітектуру KProbes. Більша частина обробки зондів виконується в контексті точки відладження та обробників винятків налагодження, які складають залежний рівень архітектури KProbes. Незалежний

від архітектури рівень KProbes - це менеджер KProbes, який використовується для реєстрації та скасування реєстрації зондів. Користувачі надають обробники зондів у модулях ядра, які реєструють зонди через менеджер KProbes[11].

JProbe повинен передати управління іншій функції, яка має той самий прототип, що і функція, на якій був розміщений зонд, а потім повернути керування вихідною функцією з тим самим станом, який був до виконання JProbe. JProbe використовує механізм, що використовується й у KProbe. Замість виклику визначеного користувачем попереднього обробника JProbe визначає власний попередній обробник, який називається setjmp_pre_handler(), і використовує інший обробник, який називається break_handler. Це триступеневий процес.

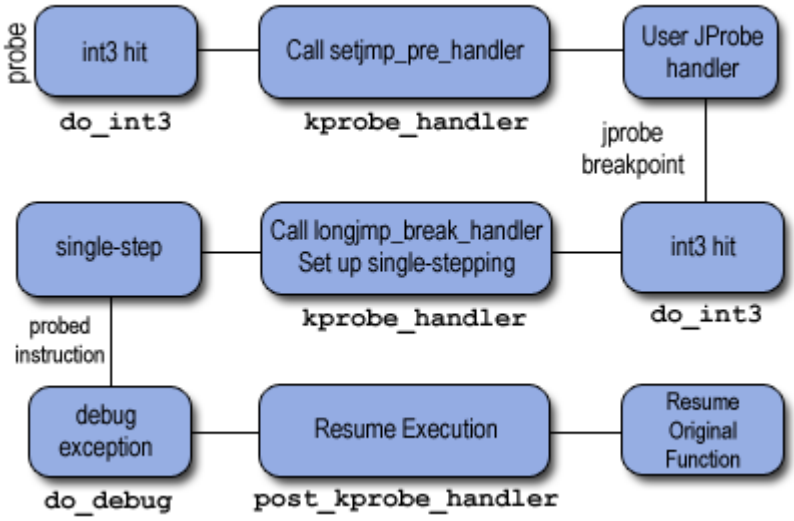

Рис. 2.19 – Принцип jprobes

На першому кроці, коли потрапляємо на точку відладження, керування досягає функції kprobe_handler(), яка у свою чергу викликає попередній обробник JProbe (setjmp_pre_handler ()). Це зберігає вміст стека та регістри перед зміною eip на адресу визначеної користувачем функції. Потім повертається значення 1, що повідомляє kprobe_handler (), що варту просто вийти, замість того, щоб налаштовувати однокрокове виконання, як для KProbe. При поверненні контроль досягає визначеної користувачем функції для доступу до аргументів вихідної

функції. Коли функція, яку визначає користувач, виконується, вона викликає jprobe_return () замість нормального повернення.

Менеджер KProbes відповідає за реєстрацію та скасування реєстрації KProbes та JProbes. Ядро файлу /kprobes.c реалізує менеджер KProbes. Кожен зонд описується структурою struct kprobe і зберігається в хеш-таблиці, хешованій адресою, за якою розміщується зонд. Доступ до цієї хеш-таблиці серіалізується за допомогою spinlock kprobe_lock. Цей фіксатор блокується перед реєстрацією нового зонда. Щоразу, коли ми потрапляємо на зонд, викликається обробник зонда з вимкненими перериваннями. Переривання відключені, оскільки обробка зонда - це багатоступінчастий процес, що включає обробку точки зупинку та однокрокове виконання зондованої інструкції. Збереження стану між цими операціями є достатньо не тривіальною, тому переривання залишаються вимкненими під час обробки зонда.

Етапи роботи з зондом залежать від архітектури; ними обробляються функції, визначені у файлі arch/kernel/kprobes.c. Після реєстрації зондів адреси, за якими вони активні, містять інструкцію точки зупинки. Як тільки виконання досягає зондованої адреси, виконується інструкція int3, в результаті чого елемент керування досягає обробника точки зупинки do_int3 () в arch/kernel/traps.c. do_int3 () викликається через шлюз переривання, тому переривання вимикаються, коли контроль надходить туди. Цей обробник повідомляє KProbes про те, що сталася точка зупинки; KProbes перевіряє, чи встановлено точку зупинки функцією реєстрації KProbes. Якщо зонд відсутній за адресою, за якою зонд було зареєстровано, він просто повертає 0. В іншому випадку викликається зареєстрована функція зонда.

KProbes - чудовий інструмент для налагодження та трасування; його також можна використовувати для вимірювання продуктивності. Розробники можуть використовувати його для відстеження шляху своїх програм всередині ядра та для

налагодження. Системні адміністратори можуть використовувати його для відстеження подій всередині ядра у виробничих системах. KProbes також можна використовувати для некритичних вимірювань продуктивності. Однак нинішня реалізація KProbes вводить певну затримку при обробці зондів. Причиною цієї затримки є єдиний kprobe_lock, який серіалізує виконання зондів на всіх процесорах на машині. Іншою причиною є механізм, що використовується KProbes, який використовує кілька винятків для обробки одного зонда. Обробка винятків - це дорога операція, яка спричиняє затримки. Потрібно провести роботу в цій галузі, щоб покращити масштабованість та покращити час обробки зонда, щоб зробити KProbes життєздатним інструментом вимірювання продуктивності.

Однак KProbes не можна використовувати безпосередньо для цих цілей. У необробленому вигляді користувач може написати модуль ядра, що реалізує обробники зондів. Однак інструменти вищого рівня необхідні, щоб зробити його зручнішим у використанні. Такі інструменти можуть містити стандартні обробники зондів, що реалізують бажані функції, або вони можуть містити засоби для створення обробників зондів, даючи їх прості описи мовою сценаріїв, наприклад DProbes.

2.7 Цілі посилення захисту

Нашим завданням є обмеження варіантів нападу, які можуть бути доступними для зловмисника. Як було описано раніше, зловмисники можуть скомпрометувати нашу операційну систему використовуючи завантажувані модулі ядра, що мають у своєму коді шкідливу для нашої системи логіку. Зловмисник може змінювати права доступу до памяті, до таблиці системних викликів, а також згубно впливати на роботу нашої операційної системи.

Як було зазначено у передньому розділі існують певні методики і принципи спрямовані на боротьбу із подібного роду програмними застосунками, проте і вони не дозволяють захистити операційну систему під керуванням ядра Linux від різного роду шкидливих завантажуваних модулів ядра. Більше того ці засоби в значній мірі украднюють успішне встановлення шкідливого програмного забезпечення у ядро, проте якщо воно все ж було встановлено, операційна система стає значно слабшою в контексті протистояння шкідливому ПО, оскільки це програмне забезпечення, потрапивши у систему може відключити усі наявні засоби захисту, за допомогою чого "відкрити двері" для усіх можливих загроз інформаційній безпеці у операційній системі Linux.

Для прикладу, для процесорів із архітектурою ARM, це можна зробити за допомогою відключення біта Never Execute bit. Якщо його значення дорівнює одиниці, то сторінки пам`яті не будуть мати змогу виконатись, не зважаючи на особливості конфігурації таблиці сторінок.

Припустимо, що зловмисник хоче змінити сторінку пам’яті з кодом, яка спочатку була доступна лише для виконання. По-перше, йому потрібно змінити біти дозволів цієї сторінки з виконання на запис. Не можливо лише добавити до файлу із правами на виконання право на запис, оскільки ці правила є взаємовиключними. Після зміни виконання на запис, зловмисник може писати на цю сторінку все що йому потрібно. Потім він знов змінює дозвіл цієї сторінки на виконання.

Альтернативою наведеним підходам до експлуатації є дублювання таблиць сторінок в іншому місці та скидання базової адреси таблиці сторінок, в ARM за це відповідає регістр TTBR. Після цього можна виконати попередню техніку. А також зловмисник може просто відключити MMU, і це обійде всі існуючі механізми захисту пам'яті, оскільки всі вони ґрунтуються на системі віртуальної пам'яті.

Виходячи з цього, в якості цілей захисту були обрані MMU та NX bit регістру SCTRL. На щастя, регістр SCTRL також контролює роботу і MMU.
Тому в даному випадку потрібно лише контролювати роботу з даним регістром.

Ці цілі були обрані в якості прикладу. Також в якості цілей посилення можна використовувати всі випадки, які оброблює kprobes.

Висновки до розділу 2

У другому розділі було розглянуто вразливості операційної системи Linux, зокрема було детально розібрано вразливість під назвою buffer overflow, було наведено приклад завантажуваного модулю ядра, який має змогу підмінювати значення у таблиці системних викликів операційної системи. Також було продемонстровано методи боротьби із такого роду взразливостями.

Було розглянуто систему примусового контролю доступу SELinux, яка реалізовано на рівні ядра. Ось кілька прикладів використання SELinux, які дозволяють побачити, яким чином можна збільшити ступінь безпеки всієї системи: створення та налагодження списку програм, які можуть читати ssh-ключі; запобігання несанкціонованого доступу до даних через mail-клієнт; налаштування браузера таким чином, щоб він міг читати в домашній папки користувача тільки необхідні для функціонування файли і папки та інші.

Було проаналізовано поняття руткіта, як вища. Визначено основні види загроз, які це програмне забезпечення може нести для операційної системи, зокрема

до таких належать: модифікція коду функцій, а також компрометація таблиці системних викликів.

Було наведено декілька існуючих зачобів боротьби із руткітами для операційної системи Linux, зокрема, такі як kprobes та jprobes. Було проаналызовано принципи роботи цих програмних засобів, а також було визначенно їх плюси і мінуси у боротьбі із завантажуваними мудулями ядра, що містяь шкідливий код.

Смисл роботи kprobes полягає в перехопленні функції та зміни її прологу таким чином, щоб можливо було перевести виконання до функції-перехоплювачу при виклику цільової функції.

Було розглянуто основні кроки, які варто зробити, аби виявити руткіти у своїй системі, а було проаналізовано, які дії варто виконати, що протистояти роботі чи встановленню руткітів у операційну систему.

Було визначено цілі посилення захисту, а саме було з`ясовано, що посилений контроль за SCTRL регістром не дасть змогу руткітам, які мають намір відключити використання модулю керуання пам`яттю (MMU), нормально функціонавати, а також дозволить виявити модулі, що містять шкідливе програмне забезпечення.

3 ПОСИЛЕННЯ ЗАХИСТУ ЯДРА LINUX

3.1 Розробка техніки посилення захисту ядра Linux від rootkit

В попередніх розділах наводилися приклади методів та технологій які використовуються для аналізу та виявлення зловмисного програмного забезпечення, але вони мають певні недоліки, які дають змогу зловмисним застосункам визначити, що вони працюють в емульованому середовищі. Тому в даній роботі пропонується інша техніка виявлення зловмисних застосунків, точніше rootkit, вона використовує технологію trustzone.

Ця техніка представляє собою аналізатор-монітор, який може збирати дані про завантажений у ядро модуль. Для того, щоб позбутися недоліків попередніх рішень, основна частина цього аналізатору буде знаходитися в swd – захищеному середовищі. Завдяки використанню довіреного середовища виконання, аналізатор працює на іншому рівні, ніж можуть виконуватися зловмисні застосунки, руткіти, а саме на рівні S-EL0. Завдяки цьому руткіти ніяк не можуть впливати на нього.

Основною ідеєю є встановлення обробників у код завантажуваних модулів без перевантаження ядра. Встановлення обробників відбувається лише у випадку, коли в коді модулю буде знайдено команди, які можуть внести зміни до регістру SCTRL. У випадку, коли їх буде знайдено замість них будуть встановлюватися обробники, в іншому – модуль буде завантажений і працювати звичайному режимі.

У розділі 1.3 було розглянуто принцип завантаження динамічного модулю у ядро Linux. Основну роботу виконує функція load_module. Вона завантажує elf заголовки і секції та виділяє під них тимчасову пам'ять. Для нас важлива секція, яка зберігає виконавчий код динамічного модулю. Її потрібно проаналізувати та знайти команду, яка працює із регістром SCTRL. Після цього, її потрібно замінити на nwd_handler (рисунок 3.1, ліва частина).

```
NWD:
kernel code {
        struct to_swd;
        do something
        .....
        if (!store_data_to_struct(to_swd) &&
            !hook(to_swd))
                return ABORT_EVENT;
        ....
}
```

```
SWD:
handler code {
        struct from_nwd;
        fill_struct(from_nwd);
        ....
        process_hook(from_nwd){
                if (check(from_nwd))
                        return OK;
                else
                        return ABORT_EVENT;
        }
}
```

Рис.3.1 – псевдокод nwd_handler, swd_handler

Nwd_handler зберігає значення регістрів цього середовища, та команду, яка намагається змінити SCTRL регістр, а після цього викликає команду SCM для переходу у режим secure monitor. Для того, щоб викликати правильний обробник swd частині в регістр r0 передається значення, яке відповідає команді обробника – TZ_MONITOR_CHECK_ NWD_SCTRL_MDF.

Для того, щоб змінити потрібні інструкції на обробник необхідно змінити права доступу до сторінки пам’яті, де зберігається вже скопійований із простору

користувача модуль (розділ 1.3), адже сторінки пам'яті ядра, що зберігають код і дані, позначені як read-only і захищені від запису.

Найпростішим рішенням даної проблеми є тимчасове відключення NX bit регістра SCTRL, але це досить небезпечне рішення, оскільки потік в якому виконується відключення захисту може також виконуватися і на інших процесорах. Щоб відключити NX біт потрібно обнулити 19 біт регістру SCTRL (рисунок 3.2):

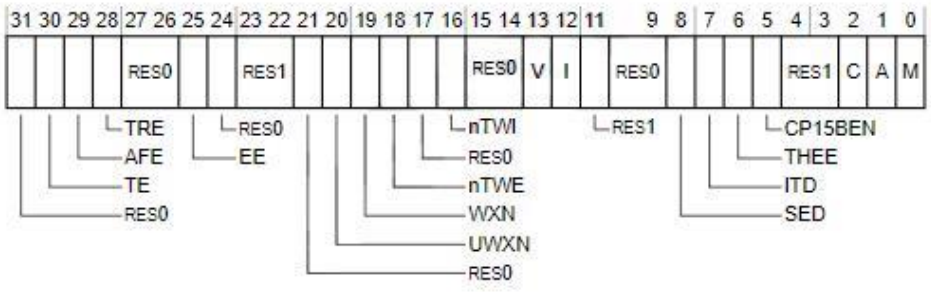

Рис. 3.2 – Регістр SCTRL

Існує також інший спосіб змінити права доступу сторінки пам’яті - створення тимчасового відображень, тому як дозволено створювати кілька посилань на фізичну пам'ять із різними атрибутами. Для цього можна використати наступний код [26]:

```
static void *map_writable(void *addr, size_t len)
{
	void *vaddr;
	int nr_pages = DIV_ROUND_UP(offset_in_page(addr) + len, PAGE_SIZE);
	struct page **pages = kmalloc(nr_pages * sizeof(*pages), GFP_KERNEL);
	void *page_addr = (void *)((unsigned long)addr & PAGE_MASK);
	int i;

	if (pages == NULL)
		return NULL;

	for (i = 0; i < nr_pages; i++) {
		if (__module_address((unsigned long)page_addr) == NULL) {
			pages[i] = virt_to_page(page_addr);
			WARN_ON(!PageReserved(pages[i]));
		} else {
			pages[i] = vmalloc_to_page(page_addr);
		}
		if (pages[i] == NULL) {
			kfree(pages);
			return NULL;
		}
		page_addr += PAGE_SIZE;
	}
	vaddr = vmap(pages, nr_pages, VM_MAP, PAGE_KERNEL);
	kfree(pages);
	if (vaddr == NULL)
		return NULL;
	return vaddr + offset_in_page(addr);
```

Рис. 3.3 – Функція для створення відображення пам’яті із доступом на запис.

Отже, після підміни команди, буде викликатися обробник, який в свою чергу буде викликати secure monitor для зміни середовища виконання.

```
push    {r4-r8, lr}
mov r8, r0
ldm r8, {r0-r7}
smc #0
stm r8, {r0-r7}
pop {r4-r8, pc}
```

Рис. 3.4 – приклад виклику secure monitor

Аргументи, які нам потрібно перевірити передаються через структуру smc_param (рисунок 3.5):

```
struct smc_param {
        uint32_t a0;
        uint32_t a1;
        uint32_t a2;
        uint32_t a3;
        uint32_t a4;
        uint32_t a5;
        uint32_t a6;
        uint32_t a7;
};
```

Рис. 3.5 – smc_param

Яка потім зчитується завдяки командам:

mov r8, r0

ldm r8, {r0-r7}

Після цього, виконавчий процес переходить до secure monitor, який передає керування на траслет, який буде виконувати обробку аргументів (рисунок 3.6).

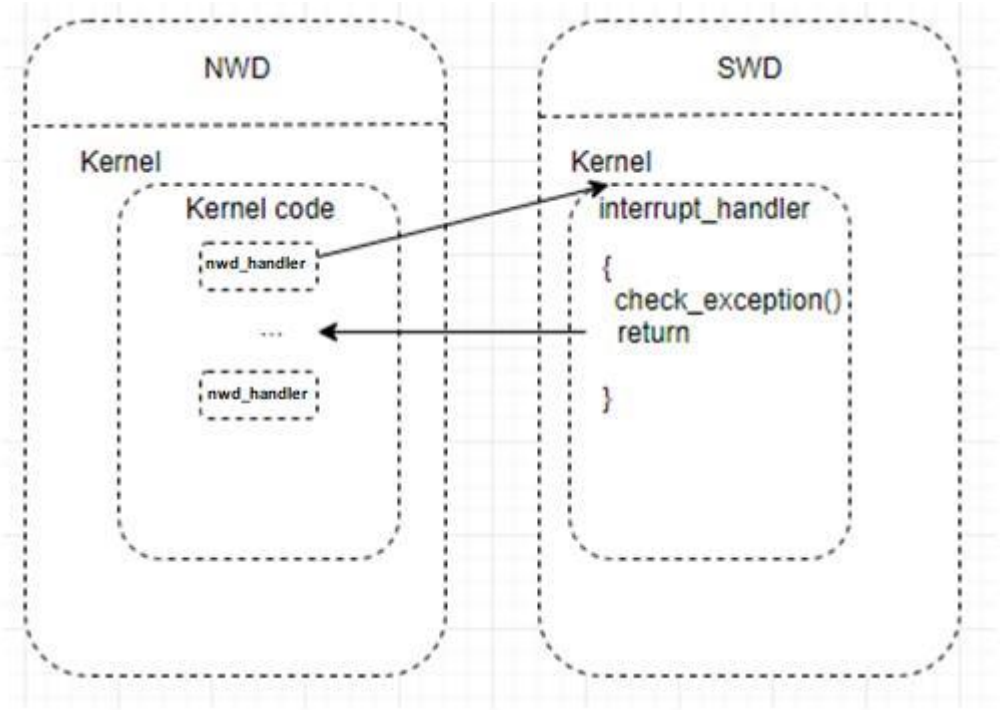

Рис. 3.6 – Роботи обробника nwd_handler

Тобто, якщо на вхід до функції load_module подається такий модуль (рисунок 3.7), наведеним алгоритмом, спочатку буде його проаналізовано на наявність інструкцій зміни SCTRL регістру, коли такі будуть знайдені (рисунок 3.8), пройде підміна цього коду на код nwd_handler (рисунок 3.4) після чого керування перейде до secure monitor, який викличе обробник в swd (рисунок 3.9).

Отже, алгоритм роботи такий:

1. Копіювання модулю із простору користувача та створення тимчасової копії.
2. Парсинг бінарного коду вхідного модулю. Під час парсингу проводиться аналіз інструкцій.
3. Якщо знайдено інструкції, які намагаються отримати доступ до регістру SCTRL, то замінити їх на інструкцію переходу в режим secure monitor, тобто SMC.
4. Завантажити та ініціалізувати досліджуваний модуль.
5. При попаданні на інструкцію SMC, обробити це виключення
6. В безпечному середовищі перевірити команду, яка намагалась отримати доступ до регістру SCTRL, якщо ці дія зловмисна заборонити її виконання, та вивантажити модуль.

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

static void disable_mmu(void)
{
        u32 fmt = 0;
        asm volatile ("mrc p15, 0, %0, c1, c0, 0" : "=r"(fmt));
}

int __init init(void)
{
        printk("Init, calling disable_mmu test\n");
        disable_mmu();
        return 0;
}

void __exit exit(void)
{
        printk("Cleanup\n");
}

module_init(init);
module_exit(exit);
MODULE_LICENSE("GPL");
```

Рис. 3.7 – Тестовий модуль

```
Disassembly of section .init.text:

00000000 <init>:
   0:   e92d4008        push    {r3, lr}
   4:   e59f000c        ldr     r0, [pc, #12]   ; 18 <init+0x18>
   8:   ebfffffe        bl      0 <printk>
   c:   ee113f10        mrc     15, 0, r3, cr1, cr0, {0}
  10:   e3a00000        mov     r0, #0
  14:   e8bd8008        pop     {r3, pc}
  18:   00000000        .word   0x00000000

Disassembly of section .exit.text:

00000000 <cleanup_module>:
```

Рис. 3.8 – Тестовий модуль на асемблері

Відповідно до SMC calling convention (розділ 1.8) у регістрі r0 передається код команди, яка повинна виконатися. У цьому випадку це буде команда TZ_MONITOR_CHECK_ NWD_SCTRL_MDF. З рисунку 3.9 можна побачити, що функція обробник-команд TA_InvokeCommandEntrypoint як раз очікує виклик цієї команди, отримавши цю команду вона викличе функцію check_sctrl_mdf, яка саме перевіряє команди із nwd частини. Якщо буде знайдено, що команда планує змінити SCTRL, то вона буде відкликана.

```
static TEE_Result check_sctrl_mdf(uint32_t param_types,
    TEE_Param params[4])
{
    uint32_t exp_param_types =
                TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_OUTPUT,
                        TEE_PARAM_TYPE_NONE,
                        TEE_PARAM_TYPE_NONE,
                        TEE_PARAM_TYPE_NONE);

    DMSG("has been called");
    if (param_types != exp_param_types)
        return TEE_ERROR_BAD_PARAMETERS;

    IMSG("Checking sctlr register");

    if (check_input_command(params[0].memref.buffer,
                            params[0].memref.size))
    {
        IMSG("Abort instruction");
        return TEE_ERROR_BAD_PARAMETERS;
    }

    return TEE_SUCCESS;
}

TEE_Result TA_InvokeCommandEntryPoint(void __maybe_unused *sess_ctx,
            uint32_t cmd_id,
            uint32_t param_types, TEE_Param params[4])
{
    (void)&sess_ctx;

    switch (cmd_id)
    {
        case TZ_TA_MONITOR_CHECK_NWD_SCTRL_MDF:
            return check_sctrl_mdf(param_types, params);
        default:
            return TEE_ERROR_BAD_PARAMETERS;
    }
}
```

Рис. 3.9 – swd_handler

3.2 Аналіз результатів

У результаті роботи були перевірені 2 наведених тестових випадки, в яких відповідно змінювалися значення регістрів, відповідних за роботу MMU та за NX bit. Спроби зміни регістру SCTRL були виявлені. Для аналізу бінарного коду для завантаження модулю необхідно використовувати вже існуючі декомпілятори, оскільки власна реалізація може мати вразливості в реалізації та погіршувати продуктивність самої системи, що приводить до підвищення часу завантаження динамічного модулю, час завантаження буде помітно вищий для великих модулів. В якості декомпілятора можливо використовувати capstone framework [27] з додатковим змінами, щоб облегшити його розміри.

Одним з недоліків є наявність false-positive помилок, оскільки можливі випадки, при яких модуль дійсно повинен змінювати значення регістру SCTRL. Цей випадок вирішується шляхом впровадження реєстрації санкціонованих модулів, та подальшої їх перевірки. Якщо модуль вже зареєстрований і є довіреним, то його можна запускати без додаткового аналізу та втручання у його бінарний код.

У порівнянні із існуючими рішення, дана технологія не має певних артефактів, які можливо виявити, як це відбувається у випадку використання емуляції. Можливо визначити модель процесору та визначити, що в даному пристрої підтримується trustzone, але вимкнути її не є тривіальною і можливою задачею.

Також потрібні налаштування та методики протидії руткітам, які можуть зашифровувати та розшифровувати свій бінарний код у runtime [28].

На відміну від jprobes, дане рішення не потребує патчу Linux Kernel image та подальшого його розповсюдження клієнтам, які потребують такого типу захисту. Оскільки дана технологія може буде включена відповідним патчем до Linux Kernel sources там бути використаною одразу після компіляції. Ін'єкція відповідних обробників відбувається динамічно лише після аналізу модулів. Якщо модуль не зареєстрований, то він потребує такого втручання.

Оскільки аналіз відбувається лише для завантажуваних модулів, а не для всього kernel image, то продуктивність системи буде вища ніж для sprobes.

Висновки до розділу 3

В даному розділі було запропоноване рішення, яке є дозволяє аналізувати динамічно завантажуванні модулі. Дане рішення використовує технологію довіреного середовища виконання, що дозволяє йому бути не доступним до руткітів.

Дане рішення можна використовувати для аналізу бінарного коду модулів, знаходячись в захищеному середовищі, не залишаючи жодних артефактів, що дають змогу руткіту упізнати аналізуючу систему.

Пропоноване рішення представляє собою певний Sandbox, в якому працює модуль. Якщо модуль звертається до важливих до інформаційної безпеки ресурсів, то замість інструкцій звернення, наприклад, до регістру SCTRL встановлюється виклик команди SMC, а далі обробник виключень буде перевіряти наведену інструкцію.

Запропоноване рішення було порівняно з типовими методами аналізу руткітів. Результат показав, що технологія TEE має ряд переваг на типовими методами. Оскільки TEE має більш привілейований доступ до ресурсів, то вона

може використовуватися як монітор, логер або аналізатор системи. Це ускладнює приховування своєї діяльності для руткітів, оскільки вони не знатимуть, що вони аналізуються.

Недоліками даної технології є складність її реалізації. Це стосується аналізу бінарного коду динамічно завантажуваного модулю. Ще одним недоліком є наявність false-positive помилок, оскільки можливі випадки, при яких модуль дійсно повинен змінювати значення регістру SCTRL. Цей випадок вирішується шляхом впровадження реєстрації санкціонованих модулів, та подальшої їх перевірки. Якщо модуль вже зареєстрований і єдовіреним, то його можна запускати без додаткового аналізу та втручання у його бінарний код.

Запропоноване рішення може використовуватися для впровадження політик роботи модулів ядра, а також може використовуватися в якості аналізатору зловмисних застосунків.

ВИСНОВКИ

У даній роботі було розглянуто особливості та основні функції операційної системи на основі ядра Linux. Було встановлено, що операційна система - це спеціальна програма, яка має повний доступ доресурсів машини, таких як процесор, операційна пам`ять та разного роду контроллери, а також вона створює певний інтерфейс взаємодії між комп`ютером та користувачем. Ядро Linux являється основною складовою операційної системи. У випадку операційних систем на базі Linux, ядро являє собою монолітний бінарний файл, який містить усь необхідну функціональність для нормального функціонування машини. Ядро Linux підтримує динамічне, тобто уже в процесі роботи операційної системи, завантаження модулів ядра. За допомогою всановлюваних модулів користувач може підключати до комп`ютера нові пристрої та успішно використовувати їх користуючись системними викликами операційної системи Linux. Операційна система імплементує системи захисту пам`яті, що ускладнюює процес проникнення та компрометацій системи.

Було розглянуто технологію під назвою Trusted Execution Environment. Основна ідея цієї технології полягає у тому, щоб розділити простори виконання на безпечний та небезпечний. Кожний із цих просторів може мати свою самостійну операційну систему для контролю свого оточення і запуску різного роду застосунків. Цінність цієї технології полягає у тому, що навіть у випадку, якщо нашаоперацийна система була скомпрометована, наші конфіденційні данні все одно залишаться неушкодженими і не потраплять у руки словмисникам. Контроль і

взаємодія між просторами виконання відбувається за допомогою спеціального цежиму процесора під назвою Secure Monitor.

Було проаналізовано загрози, які наявні на рівні ядра. Для прикладу, людина, що хоче скомпромтувати нашу систему, може скористатися однією із вразливостей у просторі користувача та отримати права суперкористувача і після цього без особливих приблем робити із нашою системою усе, що заманеться, для прикладу завантажити у ядро нашої опраційної системи руткіт.

Доволі тривіальним, але таким, який показує проблему, є приклад роботи руткіта із підміною адреса у таблиці системиних викліків. Код руткіта, який був наведений уданій роботі, може без особливих проблем дізнатись адрес таблиці системних викликів, обрати функцію, яку він хотів би перевизначити і заінити виконання вбудованої функції на самописну, яка може нести серйозну шкоду операційній системі.

Разом з цим, нападник може, отримавши необхідні права, змінювати конкретні регістри процесора під час виконання процесу. Для прикладу він може змінити регістр SCTLR для архітектури ARM, зокрема він може вимкнути біти, що відповідаються за захист нашої системи. Прикладом такого біту захисту може бути Never Execute Bit, після зміни якого зловмисник може без проблем завантажувати у оперативну память машини свій код та виконувати його. Так сомо зловмисник може відімкнути модуль керування системою пам`яті, що призведедо прямого відображення фізичних адрес із оперативної пам`яті у процеси, це дасть змогу обійти усі механізми захисту пам`яті, які імлементує архітектура процесора.

У дані роботі було запропоновано рішення, що ускладнює роботу руткітів у операційній системі або ж не дозволяє їм бути запущеними взагалі. Це рішення використовує технологію безпечного середовища, що дає змогу виявляти завантажувані модулі ядра, що поводять себе підозріло, а також унеможливлє

завантаження модулів, які намагаються змінити важливі для нормального і без печного функціонування нашої машини біти.

Підхід із використаням безпечного середовища унеможливлює виявлення аналізуючого засобу для руткітів. Запропоноване рішення, аналізує інструкції, які містяться у завантажуваному модулі ядра, і у випадку наявності там інструкцій, які змінюють значення регістрів SCTRL, відбувається перехід у режим монітора, а після цього керування передається у безпечне середовище, у якому запущений наш довірений застосунок. Наш застосунок перевірить інструкцію на предмет її шкоди нашій системі і продовжить виконання, або ж завантажння модулю, у випадку, якщо загрози не виявлено, і згенерує виключення і припинить завантаження або виконання модулю, якщо було виявлено загрозу нормальній робіті операційної системи.

Пропоноване рішення представляє собою певний Sandbox, в якому працює модуль. Якщо модуль звертається до важливих до інформаційної безпеки ресурсів, то замість інструкцій звернення, наприклад, до регістру SCTRL встановлюється виклик команди SMC, а далі обробник виключень буде перевіряти наведену інструкцію.

Було проведено аналіз існуючих засобів для боротьби з руткітами, а також було порівняно запропоноване рішення із уже наявними. Аналіз показав, що за рахунок використання у нашому рішенні безпечного середовища виконання ми маємо абсолютний доступ до ресурсів системи, завдаяки цьому дане рішення може бути використане у подальшому, як логер або ж аналізатор операційної системи на педмет руткітів.

Практична цінність роботи полягає в тому, що результати роботи можуть бути застосовані при аналізі апаратних засобів на основі операційної системи Linux на

предмет наявності у них руткітів, а також цінність полягає у можливості впровадження системи моніторингу і аналізу у апаратних сасобах на основі Linux.

ПЕРЕЛІК ПОСИЛАНЬ

1. Операционная система UNIX [Електронний ресурс]. Режим доступу до ресурсу: http://mif.vspu.ru/books/os-tutorial/unix_1.html
2. Загружеемые модули ядра Linux [Електронний ресурс]. Режим доступу до ресурсу: https://itproffi.ru/zagruzhaemye-moduli-yadra-linux/
3. Управление памятью в Linux [Електронний ресурс]. Режим доступу до ресурсу: https://itproffi.ru/upravlenie-pamyatyu-v-linux/
4. Память процесса [Електронний ресурс]. Режим доступу до ресурсу: https://parallel.uran.ru/book/export/html/536
5. Стек вызовов [Електронний ресурс]. Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Стек_вызовов
6. Introduction to Trusted Execution Environment and ARM's TrustZone [Електронний ресурс]. Режим доступу до ресурсу: https://embeddedbits.org/introduction-to-trusted-execution-environment-tee-arm-trustzone/
7. SELinux [Електронний ресурс]. Режим доступу до ресурсу: https://habr.com/ru/company/kingservers/blog/209644/
8. Rootkits [Електронний ресурс]. Режим доступу до ресурсу: https://www.anti-malware.ru/threats/rootkits
9. Rootkits introdaction [Електронний ресурс]. Режим доступу до ресурсу: http://www.compline-ufa.ru/rootkit

10. Защита от руткитов в Linux[Електронний ресурс]. Режим доступу до ресурсу: https://tech-geek.ru/linux-rootkit/
11. Защита от руткитов в Linux[Електронний ресурс]. Режим доступу до ресурсу: https://lwn.net/Articles/132196/
12. OP-TEE. Open Portable Trusted Execution Environment [Електронний ресурс]. -2016. -Режим доступу до ресурсу: https://www.op-tee.org/
13. QEMU. QEMU emulator [Електронний ресурс]. -2016. - Режим доступу до ресурсу:http://www.qemu.org/
14. ARM. MMU [Електронний ресурс]. – 2012. – Режим доступу до ресурсу: доступу до ресурсу: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500e/BABJAH
15. Ksplice. Patch Kernel Without Rebooting[Електронний ресурс] – Режим доступу до ресурсу: https://github.com/jirislaby/ksplice/blob/master/kmodsrc/ksplice.c#L1178
16. Statista. Number of Smartphone users [Електронний ресурс]. Режим доступу до ресурсу: https://www.statista.com/statistics/330695/number-of-smartphone